第9章　データとタスク並列に対する公認拡張

近代的な並列マシンは、多数のプロセッサが、それぞれ自分のデータのみをアクセスして処理を行った場合に、最高の性能に達する。それゆえ、最高性能のプログラムは、計算の分割とデータのマッピングが協調的に機能することによって得られる。3 つの公認拡張は、この調和を引き出す手段を与える。

1. ON 指示文は、並列マシンのプロセッサ間で計算を分割する (DISTRIBUTE 指示文がプロセッサ間でデータを分割するのと同じように)。
2. RESIDENT 指示文は、あるデータのアクセスを行なうのに、実装においてプロセッサ間のデータの移動が必要ないということを表明する。
3. TASK_REGION 構造は、それぞれがデータ並列 (または入れ子になったタスク並列) に実行できる、独立した粗粒度タスクを生成する手段を提供する。

3 つの構文は、この後の 9.1 節で紹介する**活動プロセッサ**の概念に関連している。プロセッサに計算を割り当てることにより、ON 指示文 (9.2 節) は活動プロセッサを定義する。RESIDENT 指示文 (9.3 節) は、そのローカリティの表明において、この活動プロセッサの集合と、マッピングの指示文によって与えられた情報を使用する。最後に、TASK_REGION 構造 (9.4 節) は、活動プロセッサ集合からそのタスクを生成する。

9.1　活動プロセッサ集合

活動プロセッサは、HPF2.0 で使用される、プロセッサとプロセッサ構成の考え方の拡張である。HPF2.0 は、プロセッサの (静的な) 集合が存在すると仮定し、プログラムは、データの配置 (たとえば、DISTRIBUTE 指示文を通して) と計算の実行 (たとえば、FORALL 文の実行によって) のためにこれらのプロセッサを使用する。プロセッサ集合をさらに分割することについてはほとんど述べられていないが、それらが役立つ場合もある (例えば、8.7 節の公認拡張におけるプロセッサ部分集合へのマッピングや、部分配列に対する計算の性能の説明)。タスク並列のような機能では、プロセッサのより動的な集合を考えることを必要とする。特に、「どのプロセッサが現在実行中か」という質問に答えるには、これらの機能を定義することが重要である。

簡単に言うと、活動プロセッサとは、HPF の文 (または文の集合) を実行するプロセッサである。活動プロセッサは、その文を実行するのに必要な全ての操作を行なう。ただし、(おそらくは) データへの最初のアクセスと結果の出力は除く。 この後で説明するように、ある種の操作は、特定のプロセッサが活動状態であることを要求するが、ほとんどの部分では、任意の文の実行において、任意のプロセッサが活動状態になることができる。HPF プログラムの実行開始時には、全てのプロセッサが活動状態である。9.2 節で述べるように、ON 指示文は、そのスコープ中の文の実行における活動プロセッサ集合を限定する。次の単純な例を考

えてみる (これは直感的な意味を持つと言える)。

```
!HPF$ ON HOME( Z(INDX) )
      X(INDX-1) = X(INDX-1) + Y(INDX) * Z(INDX+1)
```

X、Y、および Z は同じ分散で、それは複製ではないとしておく。ON 指示文に従うと、この文は次のように実行されるだろう。

1. Z(INDX) を所有するプロセッサは活動プロセッサとなる。この ON ブロックの異なる実行においては、これは異なるプロセッサかもしれない。
2. X(INDX-1)、Y(INDX)、Z(INDX+1) の値が、活動プロセッサにとって利用可能となされる。分散が同一のため、Y(INDX) はすでにそこに置かれている。その他の値を得るためには、データの分散や、プログラムを実行しているハードウェア次第で、活動プロセッサがメモリからレジスタにロードすることになるかも知れないし、あるいは 1 個か 2 個の他のプロセッサが活動プロセッサにメッセージを送ることになるかも知れない。
3. 活動プロセッサは前のステップで送られた値を使用して加算と乗算を実行する。
4. 結果は X(INDX-1) に保存されるが、これは他のプロセッサ上かもしれない。これにはやはり、同期操作や、他のプロセッサ間操作を必要とするかもしれない。

文の一つが関数またはサブルーチン呼出しを含むとき、この機構は微妙な問題を含む。9.2.4 項でこれらのケースを扱う。ON 指示文の実装に関する助言は、この後の 9.2.2 項で与えられる。

活動プロセッサの概念と関連して、用語を少し追加しておくと便利である。もし集合のすべてのプロセッサが活動状態なら、その集合は**活動プロセッサ集合**と呼ばれる。すべての活動プロセッサの集合は時々**その** (the) 活動プロセッサ集合と呼ばれる。この集合は動的で、もし文が繰り返し実行されるならその活動プロセッサ集合は毎回異なっているかもしれない。一般に、HPF 構文は活動プロセッサ集合を縮小できるだけであって、拡張はできない。しかしながら、元の活動プロセッサ集合がいくつかの独立な集合に分割されるなら、すべての分割された集合が同時に実行するかもしれない。これはまさに、TASK_REGION 構文 (9.4 節で記述) の機能である。

全体プロセッサ集合は HPF プログラムで利用可能なすべてのプロセッサの集合である。これは、主プログラムの実行が始まるときに活動状態であるプロセッサの集合と正確に同じである。

活動プロセッサ集合中にないプロセッサは**休止状態**にあると呼ばれる。(プロセッサは一つの文に関して休止状態であっても、ほかの文に関して活動状態かも知れないということに注意されたい。これは、TASK_REGION 構文の中では一般的である。)

活動プロセッサ集合の特性について問い合わせることが必要な場合もある。これは、12.1 節で記述している公認拡張の組込み関数 ACTIVE_NUM_PROCS と ACTIVE_PROCS_SHAPE によって達成される。

プロセッサにマップされたデータはその上に**レジデント** (*resident*) であるといわれる。複製された実体はそのコピーを持つすべてのプロセッサの上でレジデントである。

【仕様の根拠】 活動プロセッサ集合の縮小のみを許すことは、最初は奇妙に思うかもしれない。しかしながら、HPF の設計においてはすべてのプロセッサは実行の最初で

利用可能であると仮定する。たとえば、DISTRIBUTE の実装にはプロセッサの数 (ブロックサイズを決定するためなど) とその識別子 (メモリを割り当てて、データの移動を行うため) についての情報を必要とする。それゆえに、実行モデルは動的に分割と再結合のできるプロセッサの静的な集合を使用する。【以上】

9.1.1 SUBSET 指示文

この項では、活動状態または休止状態のプロセッサ集合と、明示的にマップされたデータとの関係について説明する。大雑把にいうと、メモリの割付けは、ローカルに行わなければならないということである。すなわち、もしプロセッサが配列の一部を保持するなら、そのプロセッサは配列が生成されたときに活動状態でなければならない。この規則は、以下のことを包含している:

- 局所的な実体は、副プログラムが呼び出されたとき又はそれらが割り付けられたときのどちらかに、活動プロセッサ集合の上に保持されなければならない。
- 仮引数は常に活動プロセッサの集合にマップされる。9.2.4 節でこれを保証するメカニズムを説明する。
- 大域的な実体 (すなわち、COMMON または MODULE 中の実体、親子結合によって参照可能になった実体) は休止状態のプロセッサに明示的にマップしてもかまわない。しかし、大域的な実体が割り付けられるとき、すなわち、(全てのプロセッサが活動状態である) プログラムの開始時、または他の副プログラムへの入口、または ALLOCATE 文の実行の時点において、それらのプロセッサは活動状態でなければならない。

局所的な実体と大域的な実体の扱いから、宣言が、2 つのクラスのプロセッサ構成を参照することがあり得るということが明らかになる。一つ目は、全体プロセッサ集合から構成され、主に大域的なデータの宣言に用いられる。これは、**全体**プロセッサ構成として知られている。これらは常に同じプロセッサを表すので、参照の固定的な枠として振舞い、矛盾の無い宣言が行なえる。*processors-directive* (規則 H329) はデフォルトで全体プロセッサ構成を定義する。活動プロセッサに適用するために、3.6 節の規則に二つのわずかな変更が必要となる：

- HPF コンパイラには、スカラであるか、または大きさ (すなわち、配列の次元の積) が全体プロセッサ集合の大きさと同じであるようなどんな**全体**プロセッサ構成でも受け入れることが要求される。
- もし二つの**全体**プロセッサ構成が同じ形状を持つなら、二つの構成の対応する要素は同じ抽象プロセッサを指すものと理解される。

どちらの場合も、変更は「全体」という言葉の追加だけである。

限定されたプロセッサ構成は、構成が宣言されるときにおいて活動状態にあるプロセッサだけを表す。それらは、局所的な実体と仮引数をマップするために使われる。部分 (subset) プロセッサ構成を宣言するには、146 ページで定義された *combined-attribute-extended* (H801) の SUBSET オプションを使用できる。あるいは SUBSET 属性の形式の文を使用することもできる。

H901 *subset-directive* **is** SUBSET *processors-name*

以下は二つの形式の例である。

```
!HPF$ PROCESSORS, SUBSET :: P(NP/4,4)
!HPF$ PROCESSORS Q(ACTIVE_NUM_PROCS())
!HPF$ SUBSET Q
```

全体プロセッサ構成の場合と同様に、部分プロセッサ構成を使用するためのいくつかの修正された規則がある：

- HPF コンパイラには、スカラであるか、または大きさが活動プロセッサの数、すなわち ACTIVE_NUM_PROCS() の呼出しによって返されるであろう数字と同じであるどんな部分プロセッサ構成も受け入れることが要求される。
- もし二つの部分プロセッサ構成が同じ形状で宣言され、その活動プロセッサ集合がそれらの宣言の間で変更されないならば、二つの構成の対応する要素は同じ抽象プロセッサを指すものと理解される。

スカラの部分プロセッサ構成は、その構成が作られたときに活動状態にあるプロセッサの一つを表すとみなされるということは重要である。

部分プロセッサ構成が、NUMBER_OF_PROCESSORS() より少ない要素を持つことが許されるということに注意されたい。これは活動プロセッサ集合が縮小できるということを反映している。二つの部分プロセッサ構成が同じであるとみなされるために条件が追加されたことにも注意されたい。これは活動プロセッサ集合の動的な特性を反映する。最後に、局所的な部分プロセッサ構成は、活動プロセッサ集合が ON 指示文によってさらに縮小されるまでは、活動プロセッサの集合を構成したものであるということに注意されたい。

9.1.2 局所的な実体と仮引数のマッピング

SAVE 属性のない局所的な実体の明示的なマッピングでは、実体の全ての要素が活動プロセッサ上にマップされるように宣言されなければならない。この要求はいくつかの場合に分けられる。

- 局所的な実体が DISTRIBUTE 指示文を通してマップされる場合、それは活動プロセッサの集合の上に分散されなければならない。一つの（しかし唯一ではない）方法は、局所的な部分プロセッサ構成を *dist-target* として使うことである。もし、明示的な ONTO 節がないなら、その実装は、*dist-target* として活動プロセッサのどんな構成を選ぶこともできる。
- 局所的な、大きさ 1 の全体プロセッサ構成は、常に活動プロセッサであると判定され、局所的な実体のための *dist-target* として現れてもかまわない。
- 局所的な実体が ALIGN 指示文によってマップされる場合、最終整列先の対応する要素は、活動プロセッサの上だけに分散されていなければならない。これはもし最終整列先の全体が活動プロセッサの上に分散されているならば確実に起こる。それはもし局所実

体が活動状態と休止状態のプロセッサの両方の上に分散されている整列先の一部に整列されており、整列された実体が「ヒットする」部分が活動プロセッサ上にのみマップされている場合にも起こる。整列によって整列先の一つ以上の次元軸に alignee が複製される場合、整列先の分散は、alignee の全てのコピーが活動プロセッサにマップされることを保証しなければならない。

いずれの場合も、活動プロセッサ集合は、DISTRIBUTE または ALIGN が有効になった時に決定する。すなわち、ALLOCATABLE 変数のためのマッピング指示文は、変数が割り付けられた時に有効になる。他の実体は、それらが宣言された時にマッピングが有効になる。

部分プロセッサ構成の宣言によりプロセッサが活動状態または休止状態になることはない。ON 指示文の実行だけがこれを行なう。特に、もしプログラムがプログラムの活動プロセッサ集合を変更する ON 指示文／構文を含んでいないなら、すべてのプロセッサは常に活動状態であり、すべての DISTRIBUTE 指示文は全体プロセッサ構成を使用できる。

明示的にマップされた大域的な実体はそれらが現れるどこででも一致したマッピングを持っていなければならない。これは通常 (COMMON 結合または USE 結合された実体に対しては)、全体プロセッサ集合への分散によって行なわれるだろう。

暗黙の (すなわち指定されていない)ONTO 節の解釈は、局所的な実体と大域的な実体によって違うことに注意されたい。大域的な実体はすべてのプロセッサの上に分散されるが、局所的な実体は活動プロセッサだけを使わなければならない。

また、全体プロセッサ構成は PROCESSORS 指示文のデフォルトであるので、活動プロセッサが導入されても、大域的な実体のマッピングへの修正は必要ないことに注意されたい。

仮引数は、上の規則を使用して、局所的な実体と同じ方法で明示的にマップされなければならない。9.2.4 項で説明されるように、この効果は、仮引数が常に活動プロセッサ集合の上に保持されるということである。ほかのデータ実体、特にその副プログラムに局所的な実体は、それゆえ仮引数に整列でき、活動プロセッサ集合に割り付けられる。

SAVE 属性を持つ実体は、それらがスコープに入るときいつでも全く同じようにマップされなければならない。それらは活動プロセッサへのマッピングの制限とは関係ない。マッピングに関しては、それらは大域的な実体と同じ規則が適用される。

9.1.3 活動プロセッサに関する他の制限

活動プロセッサ集合が全体プロセッサ集合に一致しない場合、局所的な実体と仮引数のマッピングに加えて、他のいくつかの構文が制限される。一般に、これらの制限の意図は、それが実行されるときに動作が必要とするすべてのプロセッサが活動状態であることを保証することである。特に、プロセッサにマップされたメモリの割付けや解放はそのプロセッサの協力が必要である。

REDISTRIBUTE 指示文では、活動プロセッサ集合は次のものを含まなければならない。

- REDISTRIBUTE に出会う前に *distributee* の一部分でも保持しているすべてのプロセッサ、そして
- REDISTRIBUTE が実行された後に *distributee* の一部分でも保持するプロセッサ

これは、REDISTRIBUTE 操作の前後で、再分散される実体のすべての要素が活動プロセッサに属することを意味する。実際、これは REDISTRIBUTE によるすべてのデータの移動が活動プロセッサの間で行なわれることを意味する。加えて、*distributee*(またはそれに整列しているなにか) を前もって所有しているプロセッサはメモリを解放することができ、そして *distributee* を現在所有しているプロセッサはそれのためにメモリを割り付けることができる。

同様に、REALIGN 指示文においても、活動プロセッサの集合は、REALIGN の前に *alignee* の要素が保持されているすべてのプロセッサと、REALIGN の後に *alignee* の要素が保持されるすべてのプロセッサを含まなければならない。

明示的にマップされた実体を生成する ALLOCATE 文に対しては、活動プロセッサの集合は、割り付けられた実体のためのマッピング指示文によって使用されたプロセッサを含まなければならない。割り付けられた実体の最終整列先は次の二つのクラスのどちらかに場合分けできる。

- 明示的な ONTO 句がなく分散される場合 (まったく DISTRIBUTE 指示文のない場合を含む。) この場合、コンパイラは実体を保持する活動プロセッサの集合を選ばなければならない。
- プロセッサ構成の一部に ONTO で分散される場合。この場合、指定された部分は活動プロセッサ集合の構成でなければならない。

例を挙げる。

```
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
!HPF$ ON (P(1:4))
      CALL OF_THE_WILD()
      ...

      SUBROUTINE OF_THE_WILD()
      INTEGER, ALLOCATABLE, DIMENSION(:) :: A, B, C, D, E, F
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS()), ONE_P
!HPF$ PROCESSORS, SUBSET :: Q(ACTIVE_NUM_PROCS())
!HPF$ DISTRIBUTE (BLOCK) :: A, E
!HPF$ DISTRIBUTE (BLOCK) ONTO P(1:4) :: B
!HPF$ DISTRIBUTE (*) ONTO ONE_P :: C
!HPF$ DISTRIBUTE (BLOCK) ONTO Q :: D, F

      ALLOCATE (A(100))   ! 明示的 ONTO がない; ブロックサイズはおそらく 25。
      ALLOCATE (B(100))   ! ブロックサイズは 25。
      ALLOCATE (C(100))   ! 一つの活動プロセッサで実行。
      ALLOCATE (D(100))   ! Q(1:4) で実行; ブロックサイズは 25。
!HPF$ ON HOME(B(1:50)) BEGIN
      ALLOCATE (E(100))   ! ONTO がない; E は Q(1:2) 上に割り付けられる。
      ALLOCATE (F(100))   ! 規格合致でない。Q(3:4) は休止状態だから。
!HPF$ END ON
```

明示的にマップされた実体を破壊する DEALLOCATE 文に対しては、その活動プロセッサ集合はその実体の一部の要素でも所有するすべてのプロセッサを含まなければならない。ここでも、割付けを解放される実体の最終整列先には二つの場合がある：

- プロセッサ構成の一部に分散される場合。この場合、その実体の部分が保持されているプロセッサはその割付けが解放されるときに活動状態でなければならない。これを保証する一つの方法は DEALLOCATE 文を囲んでいる ON ブロックは必ず対応する ALLOCATE も囲むようにすることである。

- 明示的な ONTO 節がなく分散される場合。(まったく DISTRIBUTE 指示文のない場合を含む。) この場合、その活動プロセッサ集合は、割付けが解放されるときに、実体の一部を保持するプロセッサが活動状態であることを保証するために、実体が割り付けられたとき活動状態であったすべてのプロセッサを含まなければならない。この場合も、もし割付けの解放を囲むすべての ON ブロックが割付け操作も囲むなら、これは保証される。

次の例が助けになるだろう。

```
      REAL, ALLOCATABLE :: X(:), Y(:)
!HPF$ PROCESSORS P(8)
!HPF$ DISTRIBUTE X(BLOCK) ONTO P(1:4)
!HPF$ DISTRIBUTE Y(CYCLIC)

!HPF$ ON ( P(1:6) )
!HPF$     ON ( P(1:5) )
              ALLOCATE( X(1000), Y(1000) )
!HPF$         ON ( P(1:3) )
                  ! Point 1
!HPF$         END ON
              ! Point 2
!HPF$     END ON
         ! Point 3
!HPF$ END ON
      ...
!HPF$ ON ( P(1:4) )
          ! Point 4
!HPF$ END ON
      ! Point 5
```

point 1 では、X と Y のどちらも解放できない。なぜなら、それらの要素を保持するいくつかのプロセッサが活動状態でない可能性があるからである。もし、一番内側の指示文が

```
!HPF$       ON ( P(1:4) )
```

であったなら X はその明示的な ONTO 節のため安全に割付けを解放することができた。しかしそれでもなお Y を解放することは正しくない。point 2 と 3 では、X と Y の両方とも安全に割付けを解放することができる。一般に、もし割付け解放が入れ子になった ON の同じレベルまたは外側のレベルにおいて起こり、そして制御の流れが外側の ON 構文から抜け出さない限り、その解放は安全である。point 4 では、ONTO 節は囲んでいる ON と一致するので X を解放することができる。しかし、Y を解放することは正しくない。なぜなら、ALLOCATE 文で活動状態であったいくつかのプロセッサ (たとえば、P(5)) は point 4 で活動状態でないからである。これは、DEALLOCATE 文が ON 句によって制御されているときに注意しなければならないことを説明している。point 5 のように、同じ手続の中の全ての ON 構造の外側で割付けの解放を実行することによっても、起こり得る問題を避けることができる。

割付け時に活動状態であり、ONTO 節でも指定されているプロセッサのうちの、一部分だけに実際の実体を分割して保持するようにもできる。

```
!HPF$ DISTRIBUTE A(BLOCK(10)) ONTO P(1:4)
      INTEGER, ALLOCATABLE :: A(:)
      ALLOCATE A(10)
!HPF$ ON (P(1))
      DEALLOCATE(A)      ! 正しい。A の全ての部分は P(1) だけが所有するため。
```

9.2 ON 指示文

ON 指示文の目的は、並列計算機上のプロセッサへの計算の分散を、プログラマが制御できるようにすることである。これは意味的には、データに対する DISTRIBUTE 指示文や ALIGN 指示文を、計算に対応させたものである。ON 指示文では、文または文の集合に対して、活動プロセッサ集合を指定することで分散が行なわれる。これにより、活動プロセッサ集合は一時的に縮小される。

もし、2 つの ON ブロックの実行での計算の間に相互関係がなければ (例えば、ON ブロックの実行が INDEPENDENT ループの 2 つの繰返しの場合)、それらの ON 指示文は、この潜在的な並列性を引き出す方法をコンパイラに対して明確に指示することになる。

9.2.1 ON 指示文の文法

ON 指示文には、単文形式と複文形式の 2 通りの記述法がある。その文法は以下の通りである。

H902 *on-directive* **is** ON *on-stuff*

H903 *on-stuff* **is** *home* [, *resident-clause*] [, *new-clause*]

H904 *on-construct* **is**
directive-origin block-on-directive
block
directive-origin end-on-directive

H905 *block-on-directive* **is** ON *on-stuff* BEGIN

H906 *end-on-directive* **is** END ON

H907 *home* **is** HOME (*variable*)
or HOME (*template-elmt*)
or (*processors-elmt*)

H908 *template-elmt* **is** *template-name* [(*section-subscript-list*)]

H909 *processors-elmt* **is** *processors-name* [(*section-subscript-list*)]

非終端記号の *resident-clause* は、9.3 節で定義される。今のところは、冒頭で述べた RESIDENT 指示文の一つの形式と考えておけば十分である。
home は、HOME というキーワードが使われない場合でも、HOME 節と呼ばれることがよくある。*variable* は、(大まかに言うと、) Fortran の文法用語で言う「配列要素、部分配列、構造型の成分などの引用」であることに注意されたい。テンプレートやプロセッサは、指示文でのみ定義されるので、*variable* には含まれない。*block* は、 Fortran の文法用語で言う「ひとかたまりで扱われる文の並び」(例えば、DO 構文の本体部) であるということにも注意されたい。

on-directive は、一種の *executable-directive* (規則 H205 参照) である。このことは、*on-directive* が実行部に現われることができるということを意味する。

on-construct は、Fortran における *executable-construct* である。このことは、このような構文が入れ子になれることを示しており、もしそうするなら、それらの構文は正しく入れ子になる。

【仕様の根拠】 *home* に関する規則 (*processors-elmt* を含む) の最後のオプションでの括弧の使い方に注意せよ。これは、次のようなあいまいさを防ぐ。

```
      INTEGER X(4)          ! X(I) はプロセッサ I 上となる。
!HPF$ PROCESSORS HOME(4)
!HPF$ DISTRIBUTE X(BLOCK)
      X = (/ 4,3,2,1 /)
!HPF$ ON HOME(X(2))
      X(2) = X(1)
```

もし括弧が必要なかったら、計算はどこで行なえばよいだろうか?

1. プロセッサ HOME(2) (すなわち、X(2) の持ち主)?
2. プロセッサ HOME(3) (すなわち、X(2) の代入前の値を使う)?
3. プロセッサ HOME(4) (すなわち、X(2) の代入後の値を使う)?

ON の定義は、明らかに解釈 1 が正しいことを示している。解釈 2 と同じ効果は、次のような指示文により得られる。

```
!HPF$ ON(HOME(X(2)))
```

解釈 3 の効果を得る方法はない。この問題に対するよい解決法として、Fortran に予約語を導入することが提案されたが、これは基盤である言語に対する変更が大きすぎるように思われた。【以上】

9.2.2 ON 指示文の意味

ON 指示文は、計算に対する活動プロセッサ集合を、*home* で指定されたプロセッサに限定する。制御される計算は、*on-directive* に対してはそのすぐ次の Fortran の文、*block-on-directive* に対しては、それに含まれる *block* である。制御される計算のことを、ON ブロックと呼ぶ。

すなわち、ON 指示文は、ON ブロックの計算を、指定されたプロセッサで行なうようにコンパイラに指示する。ALIGN や DISTRIBUTE といったマッピング指示文と同じく、これは助言であり、絶対的な命令ではない。コンパイラは、ON 指示文を無視することもできる。同じく ALIGN や DISTRIBUTE と同じように、ON 指示文は、計算の効率に影響を及ぼすが、最終的な結果に影響を及ぼすことはない。

> 【実装者への助言】 もしコンパイラがユーザの ON 指示文による助言を無視する可能性があるならば、そのコンパイラは、全ての指示文に従うことを強制するようなオプションも、ユーザに提供すべきである。仮引数と局所的な実体は、活動プロセッサ上にマップされることが要求されるため、活動プロセッサに関するユーザの助言に耳を傾けないコンパイラは、データマッピングに関するプログラマの助言のいくつかも無視することが必要となるだろう。【以上】

単文形式の ON 指示文は、それに続く最初の非注釈行に対する活動プロセッサを定義する。これを、その文に適用されるという。もしその文が複合 (compound) 文 (例えば、DO ループや IF-THEN-ELSE 構文) である場合、ON 指示文は、その中に入れ子になっている全ての文に適用される。同様に、ON 構文では、対応する END ON 指示文までの間に含まれる全ての文に対して、始めの ON 節が適用 (すなわち、活動プロセッサ集合が設定) される。

home の記述中で引用されるいかなる関数の評価も、その ON 指示文によって影響を受けることはない。これらの関数は、制御がその指示文に到達した時に活動状態である全てのプロセッサ上で呼び出される。よって、

```
!HPF$ ON HOME( P(1: (ACTIVE_NUM_PROCS() - 1)) ) ...
```

は、1 つの活動中のプロセッサを休止状態にする正しい方法であり、自己参照による矛盾は生じない。

HOME 節には、プログラム実体、テンプレート、またはプロセッサ構成を指定できる。これらの内のどれについても、1 つの要素または複数の要素が指定できる。これは、次のようにして ON ブロックを実行するプロセッサに変換される。

- HOME 節にプログラム実体が指定された場合、その実体の一部分でも所有するプロセッサ全てが、ON ブロックを実行しなければならない。例えば、A が明示的にマップされた配列である場合、

  ```
  !HPF$ ON HOME ( A(2:4) )
  ```

  は、A(2)、A(3)、及び A(4) を所有するプロセッサ上で文を実行することを、コンパイラに指示する。A が BLOCK 分散されている場合、これはおそらくひとつのプロセッサだろう。CYCLIC 分散の場合は、3 つのプロセッサになるだろう (多くのプロセッサが利用可能だと仮定して)。SHADOW 指示文 (H817) により作られた余分な要素のコピーは、HOME 節では考慮に入れられない。

- HOME 節にテンプレートの要素または部分が指定された場合、そのテンプレートの要素または部分の一部分でも所有するプロセッサ全てが、ON ブロックを実行しなければならない。上の例は、A が配列ではなくテンプレートであっても、同様にあてはまる。

- HOME 節にプロセッサ構成が指定された場合、そこで参照されたプロセッサが、ON ブロックを実行しなければならない。例えば、P がプロセッサ構成であるとすると、

```
!HPF$ ON ( P(2:4) )
```

は、その次の文を、P(2)、P(3)、P(4) の 3 つのプロセッサ上で実行する。

どの場合においても、ON 指示文は、計算を実行すべきプロセッサを指定する。より正確に言うと、9.1 節で述べられたように、ON 指示文は、それが支配する文に対して、活動プロセッサを指定する。同じ節で、ある種の文 (特に ALLOCATE と動的再マッピング指示文) が、特定のプロセッサが活動プロセッサ集合に含まれることを必要とすることについて述べられている。もし ON ブロック中にそれらの構造の内の 1 つが現われ、その活動プロセッサ集合が必要な全てのプロセッサを含まない場合、そのプログラムは規格合致ではない。

ON 指示文は、**計算**がどのようにプロセッサに分割されるのかを指定するだけであって、データ転送に関わるかもしれないプロセッサを指定するのではないことに注意されたい。また、ON 節自体は、その本体が、他のどの命令とも並列実行できることを保証するものではない。しかし、計算の場所を指定することは、データのローカリティに大きな影響をもたらす。後の例で示すように、ON と INDEPENDENT を同時に使用することにより、並列計算の負荷バランスを制御することもできる。

【実装者への助言】 HPF プログラムが Single Program Multiple Data(SPMD) コードにコンパイルされる場合、ON 節は、全てのプロセッサが、自分のプロセッサ *id* を HOME 節から生成される *id*(または *id* のリスト) と比較することによって、常に (効率は悪いにしろ) 実装できる。(他のパラダイムにおいても、同様の単純な実装方法はあり得る。) ON 節が繰り返し実行される場合、例えば DO ループの場合など、この処理を逆転させるのは有効な手段である。つまり、全てのプロセッサが全ての HOME 節のテストを行なうのではなく、コンパイラが、そのプロセッサでの HOME 節のテストが真となるようなループの繰返し範囲を求めるのである。(詳細については、9.2.3 項の「実装者への助言」を参照されたい。) 例えば、次のような複雑な場合を考えてみる。

```
DO I = 1, N
    !HPF$ ON HOME( A(MY_FCN(I)) ) BEGIN
        ...
    !HPF$ END ON
END DO
```

この場合、各プロセッサに割り当てられるループ繰返しのリストを生成するために、“inspector” (すなわち、各ループ繰返し毎に HOME 節を評価するだけのスケルトンループ) を実行するようなコードを生成できる。このリストは並列に生成することができる。

なぜなら、MY_FCN は、副作用があってはならないからである (少なくとも、プログラマは、いかなる副作用も期待できない)。しかし、*home* の計算を全てのプロセッサに分散させることは、不規則な通信パターンを必要とし、これはおそらく並列化の利点を損なうことになるだろう。一般に、より進んだコンパイラは、より複雑な HOME 節を、効率的に変換できるようになるだろう。特定のコンパイラの能力 (と限界) は、ドキュメントとしてはっきりとユーザに示されることが推奨される。

単純な実装によって「ふるい落された」プロセッサも、データ転送には参加しなければならない可能性が残されているということに注意されたい。もし、前提となるアーキテクチャが、一方向通信 (例えば、共有メモリや GET/PUT) を許している場合、このことは問題にはならない。メッセージ通信型のマシンでは、要求-応答型のプロトコルが用いられるだろう。このことにより、休止状態のプロセッサが ON ブロックの完了まで待ちループに入るか、あるいは要求を非同期に処理するような実行時システムが必要となるだろう。この場合も、特定のシステム上でどちらの場合が効率的でどちらが効率的でないかを、ドキュメントとしてプログラマに示すことが推奨される。

【以上】

【利用者への助言】 ON 指示文の *home* の様式は、かなり複雑であるかも知れない。これは両刃の剣である。すなわちそれは、非常に複雑な計算分割を表現できる。しかし、その分割の実装は、効率的ではないかも知れない。より具体的には、それは完全に負荷バランスのとれた計算を表現できるかも知れないが、コンパイラは、HOME 節を実装するために計算を逐次化しなければならないかも知れない。ON 節に対するオーバヘッドの量は、HPF コードや、コンパイラ、ハードウェアによって異なるだろうが、コンパイラには、配列のマッピングや指定されたプロセッサ構成のみに基づいて最適なコードを生成することが期待でき、*home* の複雑さが増すにつれて、生成されるコードは悪くなるだろう。ON 指示文の複雑さの大まかな指標は、それを計算するのに使われる実行時データの量である。例えば、定数のオフセットは非常に簡単であるが、順序を指定する配列は非常に複雑であるといった具合である。これに関するより具体的な例については、この後の 9.2.3 項を参照されたい。

ON 節は、DISTRIBUTE がそうであるのと同じ意味において、プログラムの意味を変更しないことに注意されたい。特に、ON 節は、それ自体は、逐次プログラムを並列プログラムに変換するものではない。なぜなら、ON ブロック内のコードは依然として、ON ブロックの外のコードとの依存があり得るからである。(別の言い方をすれば、ON はプロセスを生成しない。) 【以上】

ON 指示文の入れ子は、内側の ON 指示文で指定された活動プロセッサ集合が、外側の指示文が指示する活動プロセッサ集合に含まれるならば、可能である。*on-construct* の文法によれば、それが他の複合 (compound) 文の内側に適切に入れ子になっており、また、その内側にも複合文が適切に入れ子になることが、自動的に保証される。他の Fortran の複合文と同様に、ブロックの外側から *on-construct* の内側への制御の移動は禁止される。すなわち、*on-construct* へは、(実行可能な) ON 指示文を実行することによってのみ入ることができる。ブロック内での制御の移動は可能である。しかし、HPF では、END ON 指示文を「通り抜ける」場合を除いて、*on-construct* の内側から *on-construct* の外側への制御の移動**も**禁止され

ている。これは、普通の Fortran よりも厳しいことに注意されたい。ON 節が入れ子になっている場合、最内側の *home* が、事実上文の実行を制御する。プログラマはこれを、ON 入れ子のそれぞれのレベルが、順に後続のプロセッサ集合を制限していくと考えることができる。つまり、明らかに一番最後の制限が最も強くなければならない。見方を変えると、プログラマはこのことを、入れ子になった並列性に関する fork-join アプローチと捉えることもできる。

【仕様の根拠】 ON ブロックへの飛び込みと飛び出しに対する制御フローの制限により、ON ブロックは本質的に単一入口、単一出口の領域となる。これは、構文の意味をかなり簡潔にする。【以上】

ON 指示文が NEW 節を含む場合、その意味は INDEPENDENT 指示文における NEW 節の意味と同じになる。NEW 変数が新しく割り付けられ、ON 指示文の有効範囲に入る度に活動プロセッサに分散され、さらに ON ブロックから抜ける際に解放されても、プログラムの動作は不変である。すなわち、NEW 変数は入口では不定であり (つまり、ON ブロック内で使用される前に代入される)、出口でも不定である (つまり、ON ブロックの後では、再び代入されるまでは使用されない)。加えて NEW 変数は、ON 節の有効範囲内では、REALIGN、REDISTRIBUTE、あるいは (サブルーチン呼出し時の) 引数結合によって再マップされてはいけない。NEW 節に現われる変数がこの条件を満たさない場合、そのプログラムは HPF 規格合致ではない。9.3 節で述べられているように、NEW 変数は、どのように入れ子になった RESIDENT 指示文によって影響を受けない。

NEW 変数は、ON ブロックの入口で、暗黙の内に活動プロセッサ上で、再割付け、再マッピングが行なわれる。このため、それらを明示的にマップすることには制限がつく。

- ON ブロックの NEW 変数は、alignee として現われてはいけない
- ON ブロックの NEW 変数は、ONTO 節が無い場合に限って distributee になることができる

```
!HPF$ DISTRIBUTE X(BLOCK, *)
!HPF$ DISTRIBUTE Y ONTO P              ! 規格合致でない: ONTO 節がある。
!HPF$ ALIGN WITH X :: Z                ! 規格合致でない: ALIGN は禁止。
!HPF$ ON (P(1:4)), NEW(X, Y, Z) BEGIN
!HPF$ END ON
```

【仕様の根拠】 NEW 節を使うと、一時的な変数を簡単に作ることができる。このことは、RESIDENT 指示文が使用可能になった時に特に重要になる。これは、この後で明らかになる。【以上】

【実装者への助言】 NEW 変数は、ON ブロックの外側で使用されることはないので、ON 節の前後で一致させられている必要はない。それゆえ、ON 指示文で決定される活動プロセッサ集合の外側での通信が、実装で必要になることはない。スカラの NEW 変数は、活動プロセッサ集合上に複製するか、活動プロセッサ集合によって共有されるメモリ領域上に割り付けるべきである。ON ブロックの複数のインスタンスが同時に活動状態になる可能性があるとき、メモリは動的に割り付けなければならないことに注意されたい。これは、INDEPENDENT ループにおける NEW 変数の実装の時の要求に似ている。【以上】

9.2.3 ON 指示文の例

次の例は、ON 指示文の正しい使い方の例である。ほとんどの例は、プログラマが使いたくなりそうな慣用的な表現を示しており、作為的なものではない。簡単のため、最初のいくつかの例は、次の配列宣言を仮定している。

```
REAL A(N), B(N), C(N), D(N)
!HPF$ DISTRIBUTE A(BLOCK), B(BLOCK), C(BLOCK), D(BLOCK)
```

HPF の能力に対する要求で最もよくあるものの一つが、ループ繰返しをプロセッサに割り付ける方法を制御することである。(歴史的には、ON 節が最初に現われたのは、Kali の FORALL 構文において、まさにこの役割を果たすためだった。) これは、次の例で示すような ON 指示文により行なうことができる。

```
!HPF$ INDEPENDENT
DO I = 2, N-1
  !HPF$ ON HOME(A(I))
  A(I) = (B(I) + B(I-1) + B(I+1))/3
END DO

!HPF$ INDEPENDENT
DO J = 2, N-1
  !HPF$ ON HOME(A(J+1)) BEGIN
   A(J) = B(J+1) + C(J+1) + D(J+1)
  !HPF$ END ON
END DO
```

I ループの ON 指示文は、各ループ繰返しにおける活動プロセッサ集合を、A(I) を所有するプロセッサに設定する。言い替えればこの指示は、各プロセッサが配列 A の内の自分が所有する部分を実行するように (よって B についても同様に)、コンパイラに助言する。B(I-1) と B(I+1) の参照では、各プロセッサの最初と最後の繰返しで、外のプロセッサから取ってこなければならない (両端のプロセッサを除いて)。それらのプロセッサは HOME 節では述べられていないことに注意されたい。J ループの ON 指示文は、同様に各繰返しの活動プロセッサ集合を設定するが、コンパイラに、計算をシフトさせるよう助言する。その結果、各プロセッサは、B、C、D のベクトル和をとり、結果の一番目の要素を左隣のプロセッサに格納し、残りの結果を (1 つずつずらして)A に代入する。配列が CYCLIC 分散されている時も、指示文はやはり有効である (かつ、ローカルにないデータへのアクセスを最小にする) が、レジデントではないデータの参照は非常に多くなるだろう。

【実装者への助言】 コンパイラは、ループ本体部全体を含むただ 1 つの ON 節からなる DO ループを最も集中的に最適化することが強く推奨される。図示すると、コードは以下のようになる。

DO i = lb, ub, $stride$

```
      !HPF$ ON HOME(array(f(i))) BEGIN
        body
      !HPF$ END ON
    END DO
```

配列は何らかのマッピングをもつとする。マッピングによりプロセッサ p に与えられる要素の集合を、 myset(p) で表すことにする (例えば、BLOCK 分散の場合、myset(p) は連続した範囲の整数である)。すると、プロセッサ p 用に生成されたコードは次のようになるはずである。

```
DO i ∈ [lb : ub : stride] ∩ f^{-1}(myset(p))
    body
END DO
```

(この図では、通信や同期がどこで必要になるかは示されていない。それは、ループ本体部の解析から導き出されるべきものである。) さらに f は、恒等関数であるか、整数の係数を持つ線形関数であることが多く、そのどちらも容易に逆関数を求めることができる。これらの条件から、ループの繰返し集合を実行する技術が、最近のいくつかの会議で発表されている。【以上】

【利用者への助言】 上の I ループは、計算分割について効率良いコードを生成することを期待できる。実際、コンパイラは、各プロセッサが配列 A の自分の持つ範囲についてループ繰返しを行なうようにするだろう。J ループはもう少し複雑である。なぜなら、コンパイラは、HOME 節に記述されている関数の逆関数を求めなければならないからである。すなわち、コンパイラは A のレジデントな要素の範囲内である K に対して K=J+1 を J について解かなければならない。もちろん、この場合の解は J=K-1 である。一般に、コンパイラは、線形関数の逆関数は求めることができる。(しかし、ALIGN と DISTRIBUTE の複雑な結合は、K の表現をやっかいなものにし、それにより逆関数を求める処理にオーバヘッドがかかる可能性がある。) 【以上】

ループの一つの繰返しを分割してプロセッサ間に「ばらまく」ことは、有益なことがある。次のケースは、1 つの例を示している。

```
!HPF$ INDEPENDENT
DO I = 2, N-1
  !HPF$ ON HOME(A(I))
  A(I) = (B(I) + B(I-1) + B(I+1))/3
  !HPF$ ON HOME C(I+1)
  C(I+1) = A(I) * D(I+1)
END DO
```

この場合、ループ本体中の 2 つの文に対する活動プロセッサ集合は異なっている。最初の ON 節により、A(I) の参照は、最初の文ではレジデントである。2 つ目の ON 節により、A(I) は (ある I の値に対して) レジデントではなくなる。これは、両方の文でのデータのローカリティを最大化するが、2 つの間でデータの移動を必要とする。

【実装者への助言】 ループ中にいくつかの入れ子でない ON 節がある場合、上の方式は一般化を必要とする。本質的に、それぞれの独立した ON 節に対する繰返し範囲が求められなければならない。そして、あるプロセッサは、それらの範囲の和集合についてループ繰返しを行なう。ON 指示文によりガード されていた文は、明示的なテストによりガード されなければならない。まとめると、

```
DO i = lb, ub, stride
    !HPF$ ON HOME(array1(f1(i)))
    stmt1
    !HPF$ ON HOME(array2(f2(i)))
    stmt2
END DO
```

というコードは、プロセッサ p 上で、

$$set_1 = [lb : ub : stride] \cap {f_1}^{-1}(myset_1(p))$$

$$set_2 = [lb : ub : stride] \cap {f_2}^{-1}(myset_2(p))$$

```
DO i ∈ set1 ∪ set2
    IF (i ∈ set1) THEN
        stmt1
    END IF
    IF (i ∈ set2) THEN
        stmt2
    END IF
END DO
```

というコードになる。ここで $myset_1(p)$ は、$array_1$ 集合、$myset_2(p)$ は、$array_2$ を内部的とする要素の集合である (繰り返すが、同期と通信は別の手段によって扱われなければならない)。ループ分割 (loop distribution) やループ peeling といったコード変形は、多くの場合テストを除去するために使える。それらは、ON ブロックの間にデータ依存がある場合に特に有効である。【以上】

【利用者への助言】 このようにループ繰返しを分割することは、実行時に余分なテストが必要になったり、コンパイラによる余分な解析が必要になるだろう。コンパイラが、個々の ON 節に対して低オーバヘッドのスケジューリングを行なえたとしても、それらを結合すると必ずしも低オーバヘッドとなるとは限らない。ローカリティによる利益は、これに見合うくらいに大きいであろう。しかし、ON 節が複数であることのコストの方が大きいこともある。(ある ON ブロックが別の ON ブロックで計算された値を使用する場合、これら全てのことは特に正しく成り立つ。) 【以上】

ON 節の入れ子は自然なので、それは異なる次元の間での並列性を表現するのに役立つ。次の例を考えてみよ。

```
      REAL X(M,M)
      !HPF$ DISTRIBUTE X(BLOCK,BLOCK)

      !HPF$ INDEPENDENT, NEW(I)
      DO J = 1, M
        !HPF$ ON HOME(X(:,J)) BEGIN
          DO I = 2, M
            !HPF$ ON HOME(X(I,J))
            X(I,J) = (X(I-1,J) + X(I,J)) / 2
          END DO
        !HPF$ END ON
      END DO
```

Jループの各繰返しにおける活動プロセッサ集合は、(おそらくは全体プロセッサ集合である)プロセッサ構成の、1つの列である。Iループは、それぞれのプロセッサが自分の持つ要素について計算を行なうようにすることで、計算をさらに分割する。多くのコンパイラは、このような単純な例では、自動的にこの様な計算分割を選択するだろう。しかし、コンパイラは、外側のループを完全に並列化しようとして、内側のループを1つのプロセッサ上で逐次実行するかも知れない。(これは、非常に速い通信を持つマシンでは魅力的かもしれない。) ON節を挿入することにより、ユーザはこの戦略をとらないよう助言することになり、並列性を制限することと引き換えに、さらなるローカリティを得ることができる。ON指示文は、INDEPENDENTの表明を要求しないし、暗黙の仮定もしないことに注意されたい。両方の場合において、Iループの各繰返しは1つ前の繰返しに依存するが、ON指示文はそれでもなお計算をプロセッサ間に分割することができる。ON指示文は、ループを自動的に並列化することはない。

【実装者への助言】 上のような、「次元による」入れ子は、おそらくよくあるケースだろう。HOME節は、どのレベルにおいても、外側ループのインデックスを実行時不変であるとして扱うために、逆関数を求めることができる。【以上】

【利用者への助言】 入れ子になったON指示文は、上の例のように、HOME節がプロセッサ構成の異なる次元を参照する場合、効率の良い実装が行なわれると考えられる。これはループのレベル間での相互作用を最小にするので、実装が簡単になる。【以上】

上の例について、次のような変形を考える。

```
      !HPF$ DISTRIBUTE Y(BLOCK,*)

      !HPF$ INDEPENDENT, NEW(I)
      DO J = 1, M
        !HPF$ ON HOME(Y(:,J)) BEGIN
          DO I = 2, M
            !HPF$ ON HOME(Y(I,J))
            Y(I,J) = (Y(I-1,J) + Y(I,J)) / 2
          END DO
```

```
  !HPF$ END ON
END DO
```

ON 節が、その配列名以外は変わっていないことに注意されたい。この例に対する解釈は上の例と似ているが、外側の ON 指示文により J ループの各繰返しが全てのプロセッサに割り当てられる点が異なっている。内側の ON 指示文は、再び単純な所有者計算ルール (owner-computes rule) を指定している。プログラマは、コンパイラに、逐次計算を全てのプロセッサに分散するように指示したことになる。この場合の方が、外側ループを並列化するよりも効率の良い場合がいくつか考えられる。

1. 外側ループを並列化することは、多くのレジデントでない参照を引き起こす。なぜなら、どのプロセッサも各列の一部しか所有していないからである。レジデントでない参照が非常にコストが大きい場合 (あるいは M が比較的小さい場合)、このオーバヘッドは、並列実行による利益を上回ることがある。
2. コンパイラは、同期を挿入することを避けるため、INDEPENDENT 指示を利用するかも知れない。これは、自然なパイプライン実行を可能にする。あるプロセッサは、J のある値について I ループの自分の持つ部分を実行し、すぐに次の J の繰返しに入るだろう。よって、2 番目のプロセッサが (1 番目のプロセッサから) J=1 に対するデータを受けとっている時に、1 番目のプロセッサは J=2 の繰返しを開始するだろう。(同様のパイプラインは、上の X の例でも行なうことができる。)

明らかに、これらの ON 節が適切かどうかは、前提としている並列アーキテクチャに依存する。

> 【利用者への助言】 この例は、ON がどのようにしてソフトウェアの生産性を向上させるかについての説明となっている。X のマッピングが変われば HOME(X(I)) の「値」も変わるが、その意味するところは大抵の場合同じままである。すなわち、X に「整列して」ループを実行せよ、ということである。その上、ON 節の形式は移植性に優れており、また、違った計算分割を簡単に試してみることができる。両方の性質とも、低レベルのデータ配置の機構を覆い隠す DISTRIBUTE と ALIGN の利点に似ている。【以上】

ON 指示は、例えば間接参照の配列が使用されている場合など、コンパイラがデータのローカリティを正確に把握できない場合に特に有効である。次のような、同じループの 3 つのバリエーションを考える。

```
REAL X(N), Y(N)
INTEGER IX(M), IY(M)
!HPF$ DISTRIBUTE X(BLOCK), Y(BLOCK)
!HPF$ DISTRIBUTE IX(BLOCK), IY(BLOCK)

!HPF$ INDEPENDENT
DO I = 1, N
  !HPF$ ON HOME( X(I) )
  X(I) = Y(IX(I)) - Y(IY(I))
END DO
```

```
!HPF$ INDEPENDENT
DO J = 1, N
  !HPF$ ON HOME( IX(J) )
  X(J) = Y(IX(J)) - Y(IY(J))
END DO

!HPF$ INDEPENDENT
DO K = 1, N
  !HPF$ ON HOME( X(IX(K)) )
  X(K) = Y(IX(K)) - Y(IY(K))
END DO
```

I ループでは、各プロセッサは、配列 X の自分の持つ範囲について実行を行う。(つまり、繰返し I に対する活動プロセッサは、X(I) の持ち主である。) X(I) に対する参照のみがレジデントであることが保証される。($M \neq N$ である場合、IX と IY は X とは異なるブロックサイズを持ち、ゆえに異なったマッピングを持つ。) しかし、大体の場合において X(I) と Y(IX(I)) と Y(IY(I)) が同じプロセッサ上にあるならば、活動プロセッサに関するこの選択は、可能な中で最も良い選択だろう。(X(I) とその他の参照の内の 1 つが**常に**同じプロセッサ上にあるなら、プログラマは、9.3 節で説明される RESIDENT 節を付け加えるべきである。) 次のループでは、ループ繰返し J の活動プロセッサは、IX(J) の所有者である。IY は IX と同じ分散を持つので、IY(J) の参照は IX(J) と同じく、常にレジデントである。これは、ループ中で最も多い配列参照のタイプなので、IX と IY に特別な性質がない場合に、レジデントでないデータの参照の数を最小にする。しかし、プロセッサ間での負荷バランスは等しくないかも知れない。例えば、(N=M/2) の場合、プロセッサの半分はアイドル状態になるだろう。前の例と同じく、IX または IY の値が、Y の参照の内の 1 つが常にレジデントであることを保証するならば、RESIDENT 表明が行なわれるべきである。K ループでは、Y(IX(K)) の参照のみがレジデントであることが保証される (Y と X が同じ分散を持つため)。しかし、IX と IY の値によっては、Y(IY(K)) と X(K) が常にレジデントであることが保証されるかも知れない。3 つの REAL の値が、常にではなく、単に「だいたい」同じプロセッサにあるというだけでも、ローカリティと並列性の面で良い計算分割を与えるだろう。しかし、これらの利点は、分割を計算するコストと比較してみなければならない。HOME 節は、(おそらくは巨大な) 配列の実行時の値に依存するので、各プロセッサに割り当てるループの繰返しを決定するには相当な時間がかかるだろう。これらの議論から明らかなのは、複雑な計算分割を扱うのに魔法のような解はないということである。 多くの場合の最適な解は、アプリケーションに対する知識と、注意深いデータ構造の設計 (要素の並べ方を含む)、そして、高効率のコンパイル手法と実行時サポートを合わせたものなのである。

> 【実装者への助言】 K ループは、先に述べた inspector 技術の適用が想定される場合である。これらの例のさらに外側にループがあり、そのループが X の分散と IX の値を変更しないならば、各プロセッサでのループ繰返しの記録は、後で再利用するために保存しておくことができる。そのコストは、最悪でも配列の大きさに比例するオーダーで済む。【以上】

【利用者への助言】 現在製品化されているコンパイラで、K ループに対して低いオーバヘッドのコードを生成するようなものはないようである。前の例との違いは、HOME 節が、コンパイラによって簡単に逆関数が求められるような関数ではないからである。あるコンパイラは、実行時に HOME 節をテストしながら、全てのプロセッサ上で全ループの繰返しを実行することを選択するだろう。またあるコンパイラは、それぞれのプロセッサ毎にループ繰返しのリストを前もって計算するかも知れない。もちろん、リストを計算するにはそれなりのコストがかかる。

実際には、全ての配列を同じ大きさにすることで、上で述べたような整列の問題のいくつかは回避されるだろう。ここでの例は、説明のために書かれたものであり、良いデータ構造設計の例として書かれたものではない。【以上】

9.2.4 副プログラムの実行に対する ON 指示文

副プログラムの実行に対する ON 指示文の規則の要点は、副プログラムの実行が活動プロセッサ集合を変更しないということである。実際、呼ばれた側では、呼び側の活動プロセッサを引き継ぐ。それゆえ、

```
!HPF$ PROCESSORS P(10)
!HPF$ DISTRIBUTE X(BLOCK) ONTO P

!HPF$ ON ( P(1:3) )
      CALL PERSON_TO_PERSON()
!HPF$ ON ( P(4:7) )
      CALL COLLECT( X )
```

というコードは、PERSON_TO_PERSON を 3 つのプロセッサ上で呼出し、COLLECT を 4 つのプロセッサ上で呼び出す。COLLECT の実引数は、完全に活動プロセッサ上にあるわけではない。これは、この後で説明されるように、対応する仮引数について適切な宣言を行えば許される。

上の規則は、呼ばれた側のサブルーチンでのデータの分散と、興味深い関係がある。特に、仮引数は、局所実体と同じ制約にしたがって宣言されなければならないので、「仮」引数は、常に活動プロセッサ集合上にあることが保証される。しかし、このことは、対応する「実」引数がローカルであることを暗示するものではない。仮引数を明示的にどのようにマップすることができるか考察する。

- **指令的マッピング:** 実引数が活動プロセッサ集合にマップされていない場合、再マッピングが行なわれる。これは、指令的マッピングによって BLOCK 分散配列を CYCLIC 分散配列に再マップする場合と全く同じである。
- **記述的マッピング:** ユーザは、実引数が既に活動プロセッサ集合にマップされていることを表明している。その表明が真であれば、仮引数は既にローカルになっている。そうでない場合、コンパイラが再マッピング操作を挿入する (そして、4 節での推奨にしたがって警告を発する)。

- **転写的マッピング:** この場合は、仮引数に効率良くアクセスできるようにするため、新しい制約が課せられる。**仮引数が転写的マッピングを宣言されている場合、実引数は、副プログラムの実行の時点で活動プロセッサ集合に対してレジデントでなければならない。** これはおそらく実行時にチェックできるだろう。

結局、仮引数は常に活動プロセッサ集合上にマップされるが、実引数はその必要はない (転写的マッピングの場合を除く) ということになる。

【仕様の根拠】 局所的な実体としての仮引数の扱いは、既存の全ての Fortran(と FORTRAN) の規格に合致する。さらにそれは、プログラマの期待どおりに振舞うという利点も持つ。仮引数は、Fortran プログラムの効率を大きく引き下げるようなことがないことが期待される。これは、仮引数が常にローカルな位置に置かれることが保証されればより大きな期待を持てるようになる。また、プログラマは、引数がコピーされない「参照渡し」に慣れているかも知れない。活動プロセッサ集合へのデータマッピングに関する制約によれば、データが副プログラムの呼出し時に再マップされない限り、この実装は許される。特に述べる価値のある場合が 1 つある。それは、転写的マッピングである。プログラマが、データをそのままの場所に置いておき (INHERIT と関連の機能に一般に期待されることである)、かつ計算を実行するプロセッサを制御したい (ON の意味である) 場合、活動プロセッサ集合の基本原則 (9.1 節で導入された) によって、データは、呼出しが行なわれる前の時点でレジデントでなければならない。明示的な指示文によって再マッピングが行なわれる場合、ユーザは、再マッピングを行なうための通信コストを予期しているはずである。

これらの規則によって、ON 指示文を使わずに書かれた HPF プログラムが不当なものとなることはない。それらのプログラムにおいては、活動プロセッサ集合は決して変化しないからである (少なくとも、言語仕様から見れば)。 それゆえ、部分プロセッサ構成と全体プロセッサ構成は、交互に使用することができ、転写的マッピングの使用に関する制約は、自動的に守られる。【以上】

【実装者への助言】 これらの制約は、引数が明示的にマップされ、ON 節が使われている場合、仮引数へのアクセスに一方向通信が必要になることはないということを暗示している。もちろん、グローバルなデータへのアクセスには、大変な複雑さが残されている。もしコンパイラ自身が計算を分割する場合、ON 指示文の規則に制限されることはない。【以上】

【利用者への助言】 ON ブロックから副プログラムを呼ぶ際に覚えておかなければならないことは次のことである。すなわち、実引数が活動プロセッサ集合上に置かれているように注意せよということである。サブルーチンの引用仕様で転写的 (「何でも受け入れる」) マッピングが使用されている場合に、これが要求される。サブルーチンでその他のマッピングが使用されている場合、実引数をレジデントにしておく、データの再マッピングによるコストを回避できる**かも**しれない。(もちろん、そうすること自体は、再マッピングが起こらないということを保証するものではない。指令的な引用仕様は、BLOCK から CYCLIC への再分散を強制することもある。しかし、再マッピングは、活動プロセッサ間でのみ行なわれることが保証される。このことは、実行時システムが、より簡単で、効率の良い集合通信を行なうことを可能にする。) 【以上】

前の例に戻る。

```
!HPF$ PROCESSORS P(10)
!HPF$ DISTRIBUTE X(BLOCK) ONTO P

!HPF$ ON ( P(4:7) )
      CALL COLLECT( X )
```

もし、COLLECTが次のように宣言されていたら、

```
      SUBROUTINE COLLECT( A )
!HPF$ DISTRIBUTE A(CYCLIC)
```

呼出しは次のように行なわれる。

1. X は、10 個のプロセッサ (すなわち、P 全体) 上での BLOCK 分散から、4 つのプロセッサ (すなわち、P(4:7)) 上での CYCLIC 分散に再マップされる。これは、多対多交換のパターンとなる。
2. COLLECT は、P(4)、P(5)、P(6)、P(7) のプロセッサ上で呼び出される。サブルーチン内での A に対するアクセスは、それらのプロセッサに再分散された配列によって行なわれる。
3. A は、X の分散に戻るように再マップされる。これは、ステップ 1 の逆である。

A の分散は、4 つのプロセッサ (呼出しの内側での活動プロセッサ集合) 上であり、全体プロセッサ上ではないことに注意されたい。もし引用仕様が次のようであるとすると、

```
      SUBROUTINE COLLECT( A )
!HPF$ DISTRIBUTE A(BLOCK)
```

その処理は、再マッピングが、10 台のプロセッサ上での BLOCK 分散から 4 つのプロセッサ上での BLOCK 分散になるということを除けば、同じになる。すなわち、ブロックサイズが 2.5 倍になり (これに伴ってデータもシャッフルされる)、その後元に戻る。ここでも、A の分散は、P の全体ではなく、活動プロセッサ集合上であるということに注意されたい。

同じような例を挙げる。

```
      REAL X(100,100), Y(100,100)
!HPF$ PROCESSORS P(4), Q(2,2)
!HPF$ DISTRIBUTE X(BLOCK,*) ONTO P
!HPF$ DISTRIBUTE Y(BLOCK,BLOCK) ONTO Q

      INTERFACE
        SUBROUTINE A_CAB( B )
        REAL B(:)
!HPF$   DISTRIBUTE B *(BLOCK)
      END INTERFACE
```

```
!HPF$ ON ( P(4:7) )
      CALL A_CAB( X( 1:100, 1 ) )
!HPF$ ON HOME( X(1:100,1) )
      CALL A_CAB( X(1:100,100) )
!HPF$ ON HOME( Y(1:100,1) )
      CALL A_CAB( Y(1:100,1) )
!HPF$ ON HOME( Y(99,1:100) )
      CALL A_CAB( Y(99,1:100) )
```

この例は次のように説明される。P(4:7) 上での A_CAB(1:100,1) の呼出しにより、10 台のプロセッサから 4 台のプロセッサへの再マッピングが生じる。(この様な場合コンパイラは、4 節の説明のように、警告を生成することが期待される。) HOME(X(1:100,1)) 上での A_CAB(X(1:100,100)) の呼出しは、活動プロセッサ集合が変化しないので、そのような再マッピング (及び警告) は発生しない。それゆえ、記述的マッピングは、データが既に正しいプロセッサ上にあることを正しく表明している。最後の 2 つの例、すなわち、A_CAB(Y(1:100,1)) と A_CAB(Y(99,1:100)) の、それらの引数の HOME 上での呼出しも、再マッピング無しで行なわれる。両方の場合において、実引数は、プロセッサの部分集合 (最初の場合は Q の列、2 つ目の場合は Q の行) に、BLOCK 状に分散される。しかし、このようなより複雑な例については、コードを生成することができないコンパイラもあるかも知れない。

次の 2 つの転写的マッピングの例も重要である。

```
! モジュール内で
! PROCESSORS P(4)
! が宣言されていると仮定する
      REAL X(100)
!HPF$ DISTRIBUTE X(CYCLIC(5)) ONTO P

      INTERFACE
        SUBROUTINE FOR_HELP( C )
        REAL C(:)
!HPF$   INHERIT C
      END INTERFACE

!HPF$ ON HOME( X(11:20) )
      CALL FOR_HELP( X(11:20) )
!HPF$ ON ( P(1) )
      CALL FOR_HELP( X(51:60) )    ! 規格合致ではない
```

1 つ目の例は正しい。実引数は、活動プロセッサ集合上に (普通に) 分散される。2 つ目の例は間違いである。例えば、要素 X(51) は、P(3) 上に置かれているが、P(3) は、呼出しに対する活動プロセッサ集合に含まれていない。2 つ目の例は、ON 指示文が P(3:4) か HOME(X(11:20)) を指定していれば、正しくなる。これらは、どちらも同じプロセッサ集合を指す。

EXTRINSIC 副プログラムに対する呼出しも重要である。EXTRINSIC の呼出しに関する「標準の」HPF2.0 での記述 (6 章) は、次のように書かれている。

⇑

> 外来手続の呼出しは、通常の HPF 手続の呼出しと意味的に同じでなければならない。従って、外来手続の呼出しは、以下のように動作したかの如くに振舞わなければならない。
>
> 1. 副プログラムの呼出しの前後では、HPF の実行環境から見て厳密に同じプロセッサ集合になる。

この制約は、次のように読み替えることができる。

- 副プログラムの呼出しの前後では、HPF の実行環境から見て厳密に同じ**活動**プロセッサ集合になる。
- 副プログラムの呼出しの前後では、HPF の実行環境から見て厳密に同じ**全体**プロセッサ集合になる。

この意図するところは、元の言語の設計と同じものである。データが配置されるプロセッサは、現われることも消えることもできない。プログラムを実行しているプロセッサの集合も、プログラムに対して何の前触れもなく変化することはできない。同様に、いくつかの外来種別は、「全てのプロセッサが同期しなければならない」ことや、「各プロセッサでローカルな手続を実行する」ことを指定する。そのような言語は、「全ての**活動**プロセッサが同期しなければならない」ことや、「各**活動**プロセッサでローカルな手続を実行する」ことを意味すると解釈される。

> 【仕様の根拠】 このことは、EXTRINSIC 手続と ON 指示文の組み合わせに、並列性における fork-join モデルを提供する。これは、自然かつ意味的にもすっきりしていると思われる。【以上】

もし手続が選択戻りを用いている場合、その戻り先は、CALL 文と同じ活動プロセッサ集合を持たなくてはならない。実際には、引数として渡されるラベルは、CALL 文と同じ ON ブロック中の文を参照するものでなければならないということになる。

> 【仕様の根拠】 この制約は、ON ブロックからの飛び出しが禁止されていることに似ており、同様の理由づけがなされる。【以上】

ON 指示文中で明示的に CALL を用いることは、しばしばタスク並列性と結びつけられる。いくつかの例が、9.4 節にある。次の例は、1 次元領域分割のアルゴリズムにおいてプロセッサをどのように使用することができるかを示している。

```
!HPF$ PROCESSORS PROCS(NP)
!HPF$ DISTRIBUTE X(BLOCK) ONTO PROCS

! ILO(IP) = PROCS(IP)での下限値　を計算
! IHI(IP) = PROCS(IP)での上限値　を計算
```

```
DONE = .FALSE.
DO WHILE (.NOT. DONE)
  !HPF$ INDEPENDENT
  DO IP = 1, NP
    !HPF$ ON (PROCS(IP))
    CALL SOLVE_SUBDOMAIN( IP, X(ILO(IP):IHI(IP)) )
  END DO
  !HPF$ ON HOME(X) BEGIN
    CALL SOLVE_BOUNDARIES( X, ILO(1:NP), IHI(1:NP) )
    DONE = CONVERGENCE_TEST( X, ILO(1:NP), IHI(1:NP) )
  !HPF$ END ON
END DO
```

このアルゴリズムでは、計算領域の全体 (配列 X) が NP 個の部分領域に分けられ、それぞれが各プロセッサに割り当てられる。INDEPENDENT な IP ループは、各部分領域の内側での計算を行なう。ON 指示文は、コンパイラに、この (概念的に) 並列な計算をどのプロセッサで行なえば良いかを指示する。これは、コンパイラが他の手段で SOLVE_SUBDOMAIN でのデータのアクセスパターンを解析できない時は特に、データのローカリティを大きく高める。サブルーチン SOLVE_SUBDOMAIN は、その配列引数を、転写的マッピング又は記述的マッピングを用いて、1 つのプロセッサ上に置くことができる。次のフェーズで、プロセッサが協力して部分領域の境界を更新し、収束テストを行なう。副プログラム SOLVE_BOUNDARIES と CONVERGENCE_TEST は、IP ループと同じような、RESIDENT 節を持つループを持つかも知れない。各部分領域の上限と下限のみが記憶されることに注意されたい。これにより、異なるプロセッサは、異なる大きさの部分領域を処理することができる。しかし、各部分領域は、1 つのプロセッサでの配列 X の範囲に「ぴったりと」収まらなくてはならない。

【実装者への助言】 上の IP ループは、ブロック構造のコードを書くプログラマにとっては常套手段だろう。一般にそれは、以前に行なったように、HOME 節の逆関数を求めることにより実装することができる。ここで示されたような 1 対 1 のケース (おそらくプログラマは非常に良く使う) では、プロセッサ番号をループインデックス変数に代入し、ループの範囲を (1 回だけ) テストすることで実装できる。【以上】

【仕様の根拠】 あるコンパイラは、ON の情報を、呼出し側から呼ばれた側に、コンパイル時に、または実行時に伝搬させるかもしれない。ON 節を呼出し側と呼ばれた側で繰り返すことは、コンパイラにより良い情報を与えることになり、より良いコードの生成が行なわれるようになるだろう。【以上】

9.3 RESIDENT 節、指示文、構文

RESIDENT 節の目的は、計算によってアクセスされるデータが、活動プロセッサにマップされていることを保証することである。つまり RESIDENT は、スコープ内のある参照 (または全ての参照) により参照されるデータが、既に活動プロセッサ集合上に格納されていることを表明する。コンパイラはこの情報を、通信の発生を回避するためや、配列のアドレス計算を簡単

にするために使用できる。与えられたデータ要素がレジデントであるかどうかは、次の 2 つの事象によって決まることに注意すること。

- データがどこに格納されているか (すなわち、実体に対する DISTRUBUTE と ALIGN 属性)
- 計算がどこで実行されるか (すなわち、ON 指示文で指定される活動プロセッサ集合)

これらの理由のため、RESIDENT 節が ON 指示文に追加される。大抵の場合 ON 指示文は、これらの必要な条件が揃う一番最初の位置にある。RESIDENT 節はまた、単独の指示文としても使用可能である。これは、RESIDENT 節が表明するローカリティの情報が、ON の指示する範囲全体としては真でない場合に有効である。ON 指示文を変更することにより、ある RESIDENT 節が無効になったり、あるいはより多くの RESIDENT 節が真になるかもしれないということに注意すること。

H910 *resident-clause* **is** RESIDENT *resident-stuff*

H911 *resident-stuff* **is** [(*res-object-list*)]

H912 *resident-directive* **is** RESIDENT *resident-stuff*

H913 *resident-construct* **is**
directive-origin block-resident-directive
block
directive-origin end-resident-directive

H914 *block-resident-directive* **is** RESIDENT *resident-stuff* BEGIN

H915 *end-resident-directive* **is** END RESIDENT

H916 *res-object* **is** *object*

resident-directive は *executable-directive* の一種である。同様に、*resident-construct* は *executable-construct* の一種である。

RESIDENT 節の最上位の実体は、明示的にマップされなければならない。同様に、RESIDENT 節は、プログラム中の、活動プロセッサ集合が宣言されている場所 (すなわち、ON ブロック中) に現われなければならない。そうでない場合は、下記の表明は、コンパイラの動作についての表明となり、プログラムについての表明ではなくなる。

> 【実装者への助言】 RESIDENT は、休止状態のプロセッサが、ON 節の入口又は出口で通信する必要性をなくす。【以上】

RESIDENT 指示文は、もし計算が指定された活動プロセッサ集合によって実行されるならば、ON の範囲内でのある実体に対する参照は活動プロセッサ内で行なうことができるという、コンパイラに対しての表明である。この表明の範囲は、もし *resident-directive* 形式が使用されるなら、その次の Fortran の文であり、もし *residente-construct* 形式が使用されるなら、コード中で囲まれた *block* の範囲である。もし、RESIDENT が ON 指示文中の節として現れるなら、ON と RESIDENT は同じ文に適用される。

RESIDENT(*var*) は、*var* という字面を持つ変数が RESIDENT 指示文の有効範囲内の文に現われた場合、その変数が、活動プロセッサ集合 (すなわち、最も内側の ON 指示文によって指定されたプロセッサ集合) に存在するデータにのみアクセスするということを意味する。もし、変数 *var* が文によって参照される場合 (例えば、代入文の右辺や、条件式の評価中に現れる場合)、その実体及びその実体の全ての部分実体につき、少なくとも 1 つの複製が、活動プロセッサにマップされなければならない。もし、変数 *var* が、文によって代入される場合 (例えば、代入文の左辺や、READ 文の変数リスト中に現われる場合、及びその値を変更するおそれのある他のすべての文脈中に現れる場合)、その変数及びその変数の全ての部分実体につき、全ての複製が、活動プロセッサに属さなければならない。RESIDENT を CALL 文や関数呼出しに適用することは、この解釈をいくらか複雑にする。この問題については、第 9.3.2 項で論じられる。

RESIDENT は常に、それを囲む ON 指示文に関係した表明であることに注意すること。従って、もしコンパイラが ON 指示文を実装していない場合、RESIDENT の解釈に注意しなければならない。同様に、もしコンパイラがプログラマの指定した ALIGN や DISTRUBUTE 指示に従わない場合は、一般に RESIDENT 節を信頼することはできない。

最後に、NEW 変数は、どのような入れ子の RESIDENT 指示によっても考慮されない。詳細は以下に示されている。

> 【仕様の根拠】 変数の読み出しと書き込みが異なる扱いなのは、実装面での必要性のためである。もし、変数の値が読み込まれる場合 (書き込みはない)、それはどの正当なコピーからでも読みだすことができる。従って、RESIDENT はそれらのコピーのうち一つが利用可能であることだけを表明する。反対に、変数の値が更新される場合、複製された変数の全てのコピーは正当でなければならない。従って、RESIDENT はすべてのコピーが利用可能であることを表明する。
>
> RESIDENT 表明は、宣言されたデータマッピングと ON 節に常に関係する。なぜなら、データ参照のローカリティを決定するのに、それら両方から得られる情報が必要だからである。データマッピングはデータがどこに格納されているかを決定し、ON 節はデータがどこで使われるかを決定する。本質的には、それらはデータパスの両端の位置を決定する。RESIDENT 自体は、パスの長さがとても短いことを表明する。両端の位置を知ることなしにパスの長さを計ることができないのは明らかである。【以上】

次の例を考えてみよ。

```
!HPF$ ON HOME(Z(I)), RESIDENT(X,Y,RECORD(I))
      X(I) = Y(I+1) + RECORD(I)%FIELD1 + RECORD(I+1)%FIELD2
```

この指示文によって、次の事実が表明されている。

- HOME 指示文で使用されているため、Z(I) がもしも出現するとしたら、それはローカルである。
- RESIDENT 節のために、X(I) のすべての複製は Z(I) の複製が格納されているプロセッサに格納されている。これは、X と Z が同じマッピングをしているか、または Z が、X(I) が格納されているプロセッサ集合を含むプロセッサ集合上に複製されている場合に、真となるだろう。

- RESIDENT 節によって、Y(I+1) の少なくとも 1 つの複製が、Z(I) と同じプロセッサ上にある。これは、Y がすべてのプロセッサに複製されているか、Z(I) と Y(I+1) が、それらの配列の中で同じプロセッサにマップされた唯一の要素であるか、あるいは、

  ```
  !HPF$ ALIGN Y(J) WITH Z(J-1)
  ```

  という指示文がプログラムの他の場所に現われる場合に、真となるだろう。(他の状況により RESIDENT 表明が真となる場合もある。)

- RECORD(I) の全ての部分実体のうち少なくとも 1 つの複製が、Z(I) と同じプロセッサにマップされている。特に、RECORD(I)%FIELD1(すなわち、一つの成分からなる部分実体) の参照は、ローカルに参照可能である。このことが成り立つ状況は X(I) の場合と似ている。この例において、RECORD(I+1)%FIELD2 に関しては、どんな情報も利用できない。

もし、*res-object-list*が存在しない場合、RESIDENT 指示文の本体部の実行中における**全ての**変数に対する**全ての**参照 (ただし、それを囲む ON 指示文で NEW として宣言された変数は除く) は、上に述べた意味においてローカルである。つまり、全ての変数の値の全ての使用について、その変数の少なくとも 1 つの複製が、ON の指示するプロセッサ集合にマップされているということである。同様に、変数への代入操作全てについて、その変数の全てのコピーが ON の指示するプロセッサ集合にマップされている。NEW 変数の参照と代入は、常にレジデントだとみなされる。もし関数やサブルーチン呼出しがない場合、これが指示文の有効範囲中での全ての変数参照を指定する最も簡単な書き方である。(副プログラムの呼出しに適用された RESIDENT 節の説明は第 9.3.2 項を参照すること。) それは ALL_RESIDENT 節と名付けてもよかっただろう。しかしながら、現在の形式では、まだ指示副言語に別のキーワードは追加されていない。

もし活動プロセッサ集合が 1 つ以上のプロセッサを含むならば、RESIDENT は、変数がそれらプロセッサの内の 1 つに格納されているということを表明しているにすぎないということに注意すること。例えば、もしもある文がプロセッサ構成の一部分で実行されるならば、RESIDENT 節のある変数については、そのプロセッサ範囲内での通信が必要になるかも知れない。しかし、そのプロセッサ範囲の外側のプロセッサに対する通信は、それらの変数にとっては必要ないだろう。

【仕様の根拠】 この解釈に対する別の解釈は、RESIDENT 節で指定されたどんな変数も、全てのプロセッサにおいてローカルである、すなわち、複製されているというものである。これによりおそらくはもっと強力な最適化が可能になるだろうが、これはあまり一般的なケースではない。加えてこれは、CALL 文や複合文に適用された ON 指示文の意図に反するように思われる。例えば、

```
!HPF$ PROCESSORS PROCS(MP,MP)
!HPF$ DISTRIBUTE X(BLOCK,BLOCK) ONTO PROCS
!HPF$ ON HOME(PROCS(1,1:MP)), RESIDENT(X(K,1:N) )
      CALL FOO( X(K,1:N) )
```

という文ではおそらく、FOOがプロセッサ構成の行に沿って呼び出され、Xの要素は適切に渡されるだろう。これが現在の定義によりなされることである。もしもRESIDENTが、「全てのプロセッサ上に存在すること」を意味するのなら、その呼出しはXを複製することを強いるだろう。【以上】

プログラマが全てのプロセッサが活動中であることを知らない限り、マップされていない実体 (当然連続な実体を含む) が活動プロセッサに排他的にマップされると表明するのは正しくない。それゆえ、全体プロセッサ集合の真部分集合が活動中の時、そのような実体が *res-object-list* 中や、*res-object-list* なしのRESIDENT指示文の有効範囲内に現われることはできない。

RESIDENT指示文は、もしそれが正しければプログラムの意味を変えないと言う点で、INDEPENDENT指示文に似ている。もしRESIDENT節が正しくない場合は、そのプログラムは規格合致ではない (それゆえ、定義されない)。INDEPENDENT指示文のように、コンパイラはRESIDENT節の情報を使うかもしれないし、もしそれがコンパイラにとって不十分であれば無視するかもしれない。もしコンパイラがRESIDENT節が不正であること (すなわち、RESIDENT変数が完全に非ローカルであること) を検出できるなら、警告を発するのが妥当である。INDEPENDENT指示文とは違って、RESIDENT節の正当性は、(ON節で指定された) 計算のマッピングと (DISTRIBUTEやALIGN節で指定された) データのマッピングによる。もしコンパイラがそれらのどちらかを無視するならば、RESIDENT指示文の情報は利用することはできないだろう。

【仕様の根拠】 参照がローカルであることを知ることは、最適化のための有益な情報である。これを事実の表明として記述することは、HPFの精神に合致しており、コンパイラはそれを好きなように使うことができる。コンパイラへの助言としてそれを表現することは不都合な点を持つように思われる。この助言を別の方法で表現する幾つかの方法と、それに対する反論として次を挙げる。

- 「この参照に対する通信を生成してはならない」というのは、プログラムの意味を変える大きな可能性がある。あるプログラマはこの可能性を望んでいるが、それは「正しい指示文はプログラムの意味を変えるべきではない」というHPFの原則に反している。また、ある参照に対して一度通信を止めると、どのようにしてそれを再び行なえばよいかはっきりしない。
- 「この参照に対する通信を生成しなさい」ということは、有益な指示文ではない。なぜなら、コンパイラはとにもかくにも通信をしなければならないからである。
- 「この参照に対する通信を生成し、そしてこの場所に置きなさい」というのは有益である。なぜなら、コンパイラによるデフォルトの通信位置を変えることができるからである。それは依然としてプログラムの意味を変える可能性を持っている。それはまた、メッセージ通信と同じくらいプログラムを複雑にしてしまう可能性を秘めている。なぜなら、プログラマが通信をループの外に移そうと試みるようになるからである。

【以上】

9.3.1 RESIDENT 節の例

第 9.2.3 項で述べられた通り、ここでの我々の目的は、プログラマにとって一般に有益であるような慣用的手法を提案することである。まず、以前の 2 つの例を拡張することから始める。

RESIDENTは、コンパイラがアクセスパターンを見つけることができないような場合に最も有益である。このことは、次の例のように、間接参照の使用によってしばしば生じる。

```
REAL X(N), Y(N)
INTEGER IX1(M), IX2(M)
!HPF$ PROCESSORS P(NP)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: X, Y
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: IX, IY

!HPF$ INDEPENDENT
DO I = 1, N
  !HPF$ ON HOME( X(I) ), RESIDENT( Y(IX(I)) )
  X(I) = Y(IX(I)) - Y(IY(I))
END DO

!HPF$ INDEPENDENT
DO J = 1, N
  !HPF$ ON HOME( IX(J) ), RESIDENT( Y )
  X(J) = Y(IX(J)) - Y(IY(J))
END DO

!HPF$ INDEPENDENT
DO K = 1, N
  !HPF$ ON HOME( X(IX(K)) ), RESIDENT( X(K) )
  X(K) = Y(IX(K)) - Y(IY(K))
END DO
```

第 9.2.3 項でみたように、X(I) は I ループでいつもローカルで、IX(I) と IY(I) がローカルなのはまれである。上の RESIDENT 指示文は、Y(IX(I)) が同様にローカルであることを保証する。それはおそらく、IX を生成するアルゴリズム (例えば、全ての I について IX(I)=I である) の特性のためだろう。式 (例えば Y(IX(I))) がローカルであることは、たとえその部分式の一つ (IX(I)) がローカルでなくても可能であることに注意すること。

指示文は Y(IY(I)) についての情報を一切与えない。それはただ一つの非ローカルな値を持つかもしれないし、全ての値が非ローカルであるかもしれない。(もし非ローカルな値がないとしたら、RESIDENT 節には Y(IY(I)) も書かれているはずだと仮定する。) もしこの式によって参照されるローカル要素が多く、非ローカル要素がローカル要素から簡単に分離できるならば、コンパイラにそのことを明らかにするためにループを再構築することは価値がある。例えば、X の「最初」と「最後」の要素だけが各プロセッサ上で非ローカルであるということを知っていたと仮定する。するとループは、次のように分割できる。

```
!HPF$ INDEPENDENT, NEW(LOCALI)
DO I = 1, N
  !HPF$ ON HOME( X(I) ), RESIDENT( Y(IX(I)), Y(IY(I)) ) BEGIN
    LOCALI = MOD(I,N/NP)
    IF (LOCALI/=1 .AND. LOCALI/=0) THEN
      X(I) = Y(IX(I)) - Y(IY(I))
    END IF
  !HPF$ END ON
END DO
!HPF$ INDEPENDENT, NEW(LOCALI)
DO I = 1, NP
  !HPF$ ON (P(I)), RESIDENT( X(LOCALI), Y(IX(LOCALI)) ) BEGIN
    LOCALI = (I-1)*N/NP
    X(LOCALI) = Y(IX(LOCALI)) - Y(IY(LOCALI))
    LOCALI = I*N/NP
    X(LOCALI) = Y(IX(LOCALI)) - Y(IY(LOCALI))
  !HPF$ END ON
END DO
```

最初のループは Y(IY(I)) の局所要素を (非効率に) 処理する一方、2 番目のループは残りを (もっと効率的に) 処理する。多くのマシンでは、除算の操作を避けるために、例えば、前もって論理マスクを生成することによって、2 つのループを書き直すことは割に合うだろう。

J のループにおいて RESIDENT 節は、Y の全てのアクセスされた要素がローカルであることを表明する。この場合、これは次の表明と等価である。

```
!HPF$ RESIDENT( Y(IX(J)), Y(IY(J)) )
```

元の RESIDENT 節は、Y という字面を持つ式を参照しているだけだが、コンパイラはその部分式もまたローカルであることを推論することができる。これは、部分実体が、その「親」の実体と異なるプロセッサに存在することはできないからである。この考えによりたいてい RESIDENT 節は十分に短くすることができる。

K のループにおいて、次の参照はローカルである。

- Y(IX(K))。なぜなら、Y は X と同じ分散をしていて X(IX(K)) は (ON 節によって) ローカルであるからである。
- X(K)。これは RESIDENT 節のためである。

RESIDENT 節中に明示的に現れない場合でも、参照はローカルかもしれないことに注意すること。一つの目安として、よいコンパイラはそれらの要素を積極的に識別することだろう。

RESIDENT 節は動作の表明であるので、コンパイラは 1 つの RESIDENT 節から多くのことを推定できる。例えば、次のケースを考えてみる。

```
!HPF$ ALIGN Y(I) WITH X(I)
!HPF$ ALIGN Z(J) WITH X(J+1)

!HPF$ ON HOME( X(K) ), RESIDENT( X(INDX(K)) )
      X(K) = X(INDX(K)) + Y(INDX(K)) + Z(INDX(K))
```

コンパイラがこの代入文をコンパイルする時に次に示す仮定をすることは正当化される (コンパイラは ALIGN 指示文と ON 指示文の両方を尊重するとする)。

- X(K) は通信を必要としない (HOME 節によって)
- X(INDX(K)) は通信を必要としない (RESIDENT 節によって)
- Y(INDX(K)) は通信を必要としない (Y は X と等しいマッピングをしていて、INDX(K) は X(INDX(K)) と Y(INDX(K)) の二つの参照での使用の間に明らかに値を変えることができないため)

コンパイラは上のコードから INDX(K) や Z(INDX(K)) について、いかなる仮定をすることもできない。INDX が X に関連してどのようにマップされているかの指示がないので、ON 指示文はなんの指示も与えない。式 (ここでは、X(INDX(K))) がローカルであるということは、その部分式 (ここでは、INDX(K)) もまたローカルであるという意味ではないことに注意すること。同様に、Z のマッピングからは Z(INDX(K)) がローカルであるかどうかは決定できない。Z(INDX(K)-1) はローカルであることが示されているが、それはあまり役に立たない。もしも、コンパイラが追加の情報 (例えば、X が BLOCK で分散されていて、INDX(K) がブロックの境界付近に存在しない) を持つとしたら、もう少し推論を推し進めることができるかもしれない。

> 【実装者への助言】 良いコンパイラの一つの目安は、RESIDENT の表明を積極的に伝搬することである。これは通信コストを非常に低減するだろう。下記の「利用者への助言」中の事項に注意すること。【以上】

> 【利用者への助言】 コンパイラは、それらの推論を引き出すのにどんなに積極的かという点で異なると考えられる。高品質のコンパイラは、より多くの参照をローカルであると認識することができるだろうし、この情報をデータの移動を減らすために使うだろう。もしもある配列の要素がローカルであるならば、同じ静的なマッピングを持つ他の配列 (すなわち、共に整列されているか、大きさと DISTRIBUTE の形式が同じであるような配列) の同じ要素もまたローカルであることを、全てのコンパイラは認識すべきである。これは、どんなコンパイラも上の例にある Y(INDX(K)) をローカルとして認識すべきであるということである。配列のマッピングを動的に変更すること (すなわち、REALIGN と REDISTRIBUTE) は、そのような情報と、その情報を伝播することを制限するだろう。また、部分式を変更するおそれのある代入 (例えば、上の例においての K や INDX の全ての要素への代入) は、コンパイラの推論が保守的になることを強いるだろう。【以上】

9.3.2 RESIDENT 指示文の手続引用への適用

RESIDENT 指示文が手続引用に適用される場合、その表明はもう少し微妙になる。

- *res-object-list*が RESIDENT 指示文に現れる場合、呼び出された手続中での動作についての表明は一切行なわれない。

```
!HPF$ RESIDENT( A(I), B )
      A(I) = F( A(I), B(LO:HI) )
```

指示文は、この文で参照する全ての変数 (実引数を含む) が、現在の ON プロセッサ集合上でローカルであることを宣言している。しかし、F 自身の計算は、任意のプロセッサに格納された配列名 A と B の要素をアクセスすることもありえる。

【仕様の根拠】 ある字面を持つ要素の振舞いに関する表明を伝搬させることは、矛盾無く、有効に定義することが困難である。例えば、上の部分コードから呼ばれる次のような関数を考えてみよ。

```
REAL FUNCTION F( X, Y )
USE MODULE_DEFINING_A
REAL X, Y(:), B(I)
!HPF$ INHERIT Y
!HPF$ ALIGN B(:) WITH Y(:)
INTEGER I

Z = 0.0
DO I = 1, SIZE(Y)
    Z = Z + A(I)*X + B(I)*Y(I)
END DO
F = Z
END FUNCTION
```

A は、モジュール MODULE_DEFINING_A 中で、分散されたグローバルな配列として定義されていると仮定する。F における操作を考える上で RESIDENT 節は何を意味するべきだろうか。RESIDENT 指示文の式 A(I) は、呼出し元から見える配列 A の参照だけを合理的に意味するかもしれないし、または、A という名前の全ての配列の参照を意味するかもしれない。呼出し元での A は、ローカルであるか、F での A と同じグローバル配列 (もし呼び元が MODULE_DEFINING_A を使用していれば) であるか、違うグローバル配列 (もし呼び元が異なるモジュールを使用すれば) であるかもしれないことに注意すること。おそらく配列 B は制限範囲内である。関数 F での配列 B はローカルであり、したがって、呼出し元とは異なっている。しかし、ON 節の制約のために、ローカルな B が ON プロセッサ集合にマップされることは確かである。それゆえ、RESIDENT の表明は明らかに真である。さらに難しい問題として、RESIDENT 変数は、それらの変数と結合した仮引数になるかもしれないと

いうことがある。あいにくこれは、呼出し元の B という字面を持つ式が、F 中の Y という字面を持つ式を参照していることを意味し、それは、限界を超えて「字面上の」という定義を拡張してしまう。これら全ての理由によって、RESIDENT 節中の名前付き変数の意味を、指示文の文法上の有効範囲に制限することに決められた。【以上】

- もし RESIDENT 指示文が *res-object-list* を含まなければ、指示文は、呼出し側と**呼ばれた側の**手続中の全ての参照は、上で定義されているとおりローカルであることを表明する。例えば、次の文を考えてみよ。

```
!HPF$ RESIDENT
A(I) = F( A(I), B(LO:HI) )
```

この指示文は、この文中の全ての変数参照 (実引数を含む) が、現在の ON プロセッサ集合に対してローカルであり、F 自身はどんな非ローカル変数を参照も更新もしないことを宣言している。

【仕様の根拠】 RESIDENT の表明は、呼び出された手続に局所的なデータに対しては常に真である。これが正しい理由は、呼ばれた側の手続では宣言の ON 節を使わなければならず、それにより、明示的にマップされた局所変数を格納できるプロセッサ集合が制限されるためである。上の定義は、この表明を全ての明示的にマップされた大域的データに拡張し、とても強力な指示文を生成する。これは、ループ中で呼び出された手続における変数のアクセスについても表明を行なうという点で、INDEPENDENT 指示文の意味に似ている。RESIDENT の別の意味として、手続間で表明を伝搬することはしない (すなわち、変数並びがある場合と変数並びがない場合を同様に扱う) という意味を与えることもできただろう。しかし、このことは、あるマシンでの最適化のために十分な情報を与えるものではないと思われる。とりわけ、メッセージ通信マシンでタスク並列性を引き出すことをかなり難しくしただろう。【以上】

【実装者への助言】 変数並びのない RESIDENT は、ON プロセッサ集合の外側での一方向通信が、呼ばれた側では生成されないことを保証する。そのような手続は、実行時システムが特別な制約 (例えば、実行時システムが、全てのプロセッサが集合通信に参加することを要求する場合) を持たない限り、「活動」プロセッサ上でのみ呼び出すようにすることができる。

RESIDENT のもう一つの形式によって与えられる情報は、手続間で伝搬することも可能である。例えば、副プログラムに対する実引数が RESIDENT であると表明され、転写的に渡される場合、呼ばれた側でそれに整列したものも全て RESIDENT になる。もし情報が伝搬されなくても、その結果はあまり最適化が行なわれないだけのことだろう。【以上】

【利用者への助言】 RESIDENT の表明は手続間に渡って適用されるが、決して全てのコンパイラがこの情報を使用するというものではない。とりわけ、分割コンパイルは、起こり得る伝播を制限する。それゆえ、RESIDENT 節を呼び元の ON 指示文と呼び先の両方

に含むことはよい習慣である (この表明が正しいことを仮定しているのはもちろん当然のことである)。これにより、手続呼出しの両端をコンパイルする時に、コンパイラがRESIDENT 情報を利用可能であることが保証される。これは変数並びのない RESIDENT節に対して特に有効である。アクセスされる全てのデータがローカルであるという情報は、それがない場合には不可能であるようなたくさんの最適化を可能にする。【以上】

ローカリティの情報は、特に手続間で重要である。そこでは、*res-object-list* なしのRESIDENT 指示文は、とても有効に使うことができる。第 9.2.4 項のブロック構造の例を拡張した次の例を考えてみる。

```
!HPF$ PROCESSORS PROCS(NP)
!HPF$ DISTRIBUTE X(BLOCK) ONTO PROCS

! ILO(IP) = PROCS(IP)での下限値　を計算
! IHI(IP) = PROCS(IP)での上限値　を計算
DONE = .FALSE.
DO WHILE (.NOT. DONE)
  !HPF$ INDEPENDENT
  DO IP = 1, NP
    !HPF$ ON (PROCS(IP)), RESIDENT
    CALL SOLVE_SUBDOMAIN( IP, X(ILO(IP):IHI(IP)) )
  END DO
  !HPF$ ON HOME(X) BEGIN
    CALL SOLVE_BOUNDARIES( X, ILO(1:NP), IHI(1:NP) )
    !HPF$ RESIDENT
    DONE = CONVERGENCE_TEST( X, ILO(1:NP), IHI(1:NP) )
  !HPF$ END ON
END DO
```

INDEPENDENT IP ループがそれぞれの部分領域の内側の計算を行ない、各部分領域は特定のプロセッサにマップされているということを思いだして欲しい。最初の RESIDENT 節は、コンパイラに、各部分領域が他のプロセッサのデータを使わないという情報を追加する。この情報なしでは、コンパイラは最悪の場合の筋書きを想定しなければならないだろう。すなわち、それぞれの部分領域が、ローカルでない読み込み専用のデータに基づいて更新を行なうということである。どんな非ローカルデータも、INDEPENDENT 指示文に反することなく他のプロセッサによって書き込まれるということはできない。しかし、もしデータが更新されないなら (例えば、巨大な検索テーブルとか)、リモートのプロセッサに格納することは可能である。特に非共有メモリマシンにおいては、このリモートデータのアクセスは難しいだろう。RESIDENT節は、この可能性が考慮される必要のないことを保証する。SOLVE_SUBDOMAIN が必要とする全てのデータはローカルである。2 番目の RESIDENT 節は CONVERGENCE_TEST にとっての全てのデータは、X と同じプロセッサに格納されていることを表明する。同じことは、RESIDENT指示文の有効範囲中にない SOLVE_BOUNDARIES に対してはあてはまらない。例えば、必要なデータが PROCS 以外のプロセッサ構成に存在するかも知れない。このデータをアクセスすることは、上に記述されたように計算におけるボトルネックとなる可能性がある。

【利用者への助言】ここでも、コンパイラに情報を与えるというRESIDENT節の有効性に注意すること。ILOとIHIの明らかでない代入を解明することのできるコンパイラはほとんど存在しないだろう。現在のコンパイラは、上のコード中の注釈に書いてあるようなことを理解しようとさえしないだろう。【以上】

9.4 TASK_REGION構文

HPFにおけるタスク並列性は、部分プロセッサへデータ実体をマップし、異なるコードブロックを異なる部分プロセッサ上で同時実行することを許す表明を追加することにより表現される。データ実体は、プロセッサ構成のある部分上に分散されることにより、部分プロセッサにマップされる。部分プロセッサ上での実行は、ON指示文により指定される。この節では、独立した部分プロセッサ上でコードブロックを並列に実行することを指定するためのTASK_REGION指示文を導入する。

TASK_REGION指示文は、コードブロックが以下の制約を満たすことを表明するために使用される。あるタスク範囲の内側の、ソース上で最外側のONブロックは、RESIDENT属性を持たなければならない。それは、ONブロック内でアクセスされる全てのデータが、対応する活動プロセッサ集合にマップされていることを意味する。さらに、前述の条件を満たす2つのONブロックの内側のコードは、互いに干渉するI/Oを持ってはいけない。これらの制約の元では、そのような2つのONブロックは、もし独立した部分プロセッサ上で実行するなら、安全に並列実行することができる。

9.4.1 TASK_REGION構造の文法

タスク範囲とは、一対の注釈によって区切られる単一の入口を持つ範囲である。

H917 *task-region-construct* **is**
directive-origin block-task-region-directive
block
directive-origin end-task-region-directive

H918 *block-task-region-directive* **is** TASK_REGION

H919 *end-task-region-directive* **is** END TASK_REGION

*task-region-construct*は、*executable-construct*の一種である。

*task-region-construct*の外側から*task-region-construct*の内側への制御の移行があってはいけない。*task-region-construct*の外側への移行は、その移行がONブロックの内側からでないならば許される(その理由は後で明らかになる)。

9.4.2 TASK_REGION構文の意味

TASK_REGION ... END TASK_REGIONの対によって囲まれたコードブロックを、タスク範囲と呼ぶ。TASK_REGION指示はプログラマにとって、あるコードの範囲が一定の条件を満たしていることを表明するための手段となる。コンパイラは、それらの表明に従ってタスク並列コードを生成することが期待される。

タスク範囲は、ON 部分プロセッサで実行することを指定されたコードブロックを含むことができる。その他のコードは、全ての活動プロセッサを含む部分プロセッサ上で実行される。あるタスク範囲の内側にあり、入れ子のレベルが最外側の ON ブロック (すなわち、他の ON ブロックや他のタスク範囲の内側にない) は全て、**字面上のタスク** (*lexical task*) として定義される。字面上のタスクの実行の全てのインスタンスは、**実行タスク** (*execution task*) として定義される。これは、文脈から明らかに区別できる場合は、単に**タスク**と呼ばれることもある。実行タスクは、この章の始めの方で説明した**活動プロセッサ**と関連付けられる。

タスク範囲の内側では、以下の制限が守られなければならない。

- 字面上のタスクに対応する全ての ON ブロックは、RESIDENT 属性を持たなければならない。これは、実行タスクの内側での変数の読み出しにおいて、対応する活動プロセッサは、その変数の少なくとも一つのコピーを持っていなければならないことを、また、書き込みにおいては、その変数の全てのコピーを持っていなければならないことを意味する。

- 実行タスク内の入出力操作が、他の実行タスク内の入出力と干渉することは、2 つのタスクが同一の部分プロセッサ上で実行されるときのみ許される。その 2 つの実行タスクは、同じ、あるいは異なる字面のタスクのインスタンスとなれることに注意すること。一般に、2 つの入出力は、同一ファイルあるいは同一装置にアクセスする時に干渉する。2 つの入出力の干渉の条件は、第 5.1 節、INDEPENDENT 指示において詳細に記述されている。

9.4.3 実行モデルと使用法

タスク範囲により、根本的に新しい実行モデルが導入されるわけではない。しかし、タスク範囲が暗示する表明は、実行タスクで活動プロセッサとして指定されたもののみがそのタスクの実行に関与する必要があり、その他の活動プロセッサはその実行をスキップすることができるということを示している。タスク範囲を実行するプロセッサは、それが属する部分プロセッサ上で実行される全てのタスクの実行に参加し、それが属さない部分プロセッサ上で実行されるタスクの実行には参加しない。字面上のタスクの外側のコードは、そのタスク範囲の全ての活動プロセッサによって、通常のデータ並列コードとして実行される。タスク範囲に対するアクセス制限は、この実行パラダイムにより得られる結果が、タスク範囲を純粋にデータ並列実行した時に得られる結果と矛盾しないことを保証する。

タスク範囲は、タスク並列とデータ並列が統合されたプログラムを書くための、簡単だが強力なモデルを提供する。ここでは、3 つの基本的な計算構造を説明し、このモデルによってタスク並列性が効率良く引き出せることを示す。

1. 並列セクション: タスク範囲は、利用できるプロセッサを、計算を行なうための独立な集合に分け、**並列セクション**と呼ばれるものをシミュレートするために用いられる。この形式のタスク並列性は比較的率直で、かなりのアプリケーション、例えばマルチブロックアプリケーションにおいて役に立つ。タスク範囲は、独立した部分プロセッサに対する一連の RESIDENT ON ブロックを含むだけである。プロセッサを部分プロセッサに分割することは、動的に行なえることに注意すること。すなわち、実行中に計算された他の変数によって分割することが可能である。

2. 入れ子になった並列性: タスク範囲は入れ子にできる。特に、ある実行タスクでのサブルーチン呼出しは、他のタスク範囲指示を用いてさらに活動プロセッサを分割できる。これは、入れ子になった並列性の利用を可能にする。例えば、動的な木構造の分割統治計算があげられる。具体的な例として、クイックソートは、ピボットを中心にして入力キー配列を再帰的に分割し、分割の結果得られる新しい 2 つの配列に、それに比例した数のプロセッサを割り当てることにより実装できる。
3. データ並列パイプライン: タスク範囲は、パイプライン化されたデータの並列計算に利用することができる。これを 2 次元高速フーリエ変換 (2DFFT) 計算で説明する。2D FFT の初期段階では、2 次元行列を読み込み、その行列の各列について 1 次元 FFT を行なう。第 2 段階では、行列の各行に対して 1 次元 FFT を行い、最終出力を行なう。この形式の 2D FFT のパイプライン化されたデータ並列の実装では、二つの段階は、互いに重ならない二つの部分プロセッサ上にマップされる。2D FFT のタスク並列化とデータ並列化を併用したコードは、第 9.4.5 項で詳細に記述される。

9.4.4 実装

タスク範囲は、コードブロックに対する単なる表明であり、タスク並列性の利用は、少なくとも部分的にはコンパイラ構成に依存する。一方、どのようにタスク並列化が利用されるかについての詳細は、並列システム機構、コンパイラ、用いている通信モデルに強く依存する。ここでは、ある重要な考慮点を指摘し、例を用いてタスク並列化コード生成を説明する。主に、メッセージパッシングによる通信と同期を用いた分散メモリ型マシンを扱うが、共有メモリの実装に関係する幾つかの重要な議論も行う。

9.4.4.1 ローカル化された計算と通信

まず重要なことは、実行タスク中の計算と通信は、適切な `ON` 節において実行することを指定されたプロセッサ以外を含んではならないということである。

字面上のタスクの入口において、コンパイラは、非活動プロセッサがタスクの直後のコードに飛んで行くように、チェックを挿入しなければならない。実行タスクは活動プロセッサ集合の外側のデータにはアクセスできないので、関連する活動プロセッサと他のプロセッサの間には通信を生成する必要はない。メッセージ通信モデルにおいて、必要なメッセージのみ生成する通信生成アルゴリズムでは、自然と要求された結果にたどり着くだろう。しかし、通信しないプロセッサ間での空のメッセージを生成するようなことがあり得るような通信手段では、実行タスクの活動プロセッサと他のプロセッサ間で空のメッセージが生成されないようにしなければならない。

バリア同期を用いる通信モデル (共有、分散メモリにおいて) では、実行タスク内の全てのバリアは、活動プロセッサの範囲内のみでの部分バリアでなければならない。実装では、実行タスクの内側と外側のデータアクセスの一致のために、実行タスクの出入口での部分バリアを含むことが必要となるかもしれない。一般に、翻訳の枠組は、実行タスクの内側と外側のデータアクセスの一致を保たなければならない。それは、共有あるいは分散メモリ環境での仮想的な同期機構の枠組を用いて行なうことができる。

9.4.4.2 複製された計算

複製された変数だけを含む全ての計算は、実行中のプロセッサ全てに複製されるべきである。これに代わる簡単な方法として、一つのプロセッサが計算を実行し、その結果を全てのプロセッサにブロードキャストする方法がある。複製は、HPF において一般に有効だが、タスク範囲では重要な問題が加わる。というのは、ブロードキャストによって生成される通信により余分な同期が必要となり、タスク並列性の妨げになるかもしれないからである。

9.4.4.3 入出力の持つ意味

幾つかの並列システムの実装では、入出力はシステム上の一つのプロセッサを介して行われる。入出力が存在する場合のタスク並列性では、全てのプロセッサが独立に入出力できることを想定している。このパラダイムはサポートされるべきであるが、各プロセッサは、物理的に全ての入出力処理を独立に実行できなければならないわけではない。一つの簡単な答えは、全ての入出力を行なうが、実行プロセッサとはみなされない、一つの入出力用プロセッサを持つことである。そうすれば、そのプロセッサは実行に関係する依存を持たないだろう。

9.4.4.4 SPMD コード 生成か MIMD コード 生成か

コンパイラに関するもう一つの議論は、同一コードイメージを全てのプロセッサで実行するべきかどうかと言うことである。異なるプロセッサグループは異なる変数を必要とするかもしれないので、単純な SPMD 実装では、メモリを浪費するかも知れない。なぜなら、全てのプロセッサ上の全ての変数を割り付けなければならないからである。動的メモリ割付けを使うこともできるが、複雑さも増す。異なる部分プロセッサでは異なるコードイメージを使うというのも一つの解だが、これも複雑さの増加を招くだろう。

9.4.5 例: 2-D FFT

この節では、タスク並列性を用いて、パイプライン化されたデータ並列 2 次元 FFT を構築する方法を示し、HPF プログラムから生成される SPMD コードを示すことで、タスク並列がどのようにコンパイルされるのかを説明する。

基本となる逐次版の 2D FFT コードを以下に示す:

```
      REAL, DIMENSION(n,n) :: a1, a2

      DO WHILE(.true.)
          READ (unit = 1, end = 100) a1
          CALL rowffts(a1)
          a2 = a1
          CALL colffts(a2)
          WRITE (unit = 2) a2
          CYCLE
100       CONTINUE
          EXIT
```

```
        END DO
```

パイプライン化された、タスク及びデータ並列の 2D FFT を HPF で記述するためには、コードにわずかな修正と、幾つかの HPF 指示文が必要である。まず第 1 に、a1 と a2 は、別々の部分プロセッサ上に分散され、rowffts と colffts を異なる部分プロセッサ上で実行するために、タスク範囲で 2 つの字面上のタスクを生成する。タスク範囲内の代入文 a2=a1 は、タスク間のデータの転送を指定する。新しい変数 done1 は、終了条件を設定するために導入する。修正したコードは以下のようになる:

```
        REAL, DIMENSION(n,n) :: a1,a2
        LOGICAL done1
!HPF$   PROCESSORS procs(8)

!HPF$   DISTRIBUTE a1(block,*) ONTO procs(1:4)
!HPF$   DISTRIBUTE a2(*,block) ONTO procs(5:8)

!HPF$   TEMPLATE, DIMENSION(4), DISTRIBUTE(BLOCK) ONTO procs(1:4) :: td1
!HPF$   ALIGN WITH td1(*) :: done1

!HPF$   TASK_REGION
        done1 = .false.
        DO WHILE (.true.)
!HPF$       ON (procs(1:4)), RESIDENT BEGIN
              READ (unit = iu,end=100) a1
              CALL rowffts(a1)
              GOTO 101
    100       done1 = .true.
    101       CONTINUE
!HPF$       END ON

            IF (done1) EXIT
            a2 = a1

!HPF$       ON (procs(5:8)), RESIDENT BEGIN
               CALL colffts(a2)
               WRITE(unit = ou) a2
!HPF$       END ON
        END DO
!HPF$   END TASK_REGION
```

最後に、それぞれのプロセッサ用に生成された簡単化した SPMD コードを示す。ここでは、送信は非同期かつノンブロッキング、受信はデータが到着するまでブロックするようなメッセージ通信モデルを仮定している。また、ある変数がプロセッサの部分集合上のみで参

照される場合でも、全てのプロセッサで同一の宣言をするような、簡単なメモリモデルを用いている。シャドウ変数 done1_copy は、部分プロセッサ 1 から部分プロセッサ 2 に処理の終了を知らせる情報を転送するために、コンパイラによって生成された変数である。そのコードは下記のようになる:

```
          REAL DIMENSION(n/4,n) :: a1
          REAL DIMENSION(n,n/4) :: a2
          LOGICAL done1

  C       以下はコンパイラにより生成された変数
          LOGICAL done1_copy
          LOGICAL inset1, inset2
  C
  C       次の関数呼出しは、subset 1 と subset 2 のそれぞれのプロセッサに
  C       対し、変数 inset1 と inset2 を .true. に設定し、それ以外に対しては
  C       .false. を設定する、コンパイラのおまじないである
  C
          CALL initialize_tasksets(inset1,inset2)

  C       部分プロセッサ 1 用のコード
          IF (inset1)
             done1 = .false.
             DO WHILE (.true.)

  C       READ のコードは I/O モデルに依存するので変更せずに残しておく
                READ (unit = 1,end=100) a1

                CALL rowffts(a1)
                GOTO 101
      100       done1 = .true.
      101       CONTINUE
                _send(done1,procs(5:8))
                IF (done1) EXIT
                _send(a1,proces(5:8))
             END DO
           END IF

  C       部分プロセッサ 2 用のコード
          IF (inset2)
             DO WHILE(.true.)
                _receive(done1_copy,procs(1:4))
                IF (local_done1) EXIT
                _receive(a2,procs(1:4))
```

```
            CALL colffts(a2)

C       WRITEのコードはI/Oモデルに依存するので変更せずに残しておく
            WRITE (unit = 2) a2
          END DO
        END IF
```

_sendと_receiveは、部分プロセッサ間で変数を転送するための通信呼出しである。入力終了までのプログラムの実行は以下のようになる。部分プロセッサ1は繰返し入力を読み込み、rowfftsを計算し、計算結果と、通常は.false.であるフラグdone1を、部分プロセッサ2に送る。部分プロセッサ2は、フラグとデータセットを受け取り、colfftsを計算し、計算結果を出力する。入力が終了したとき、部分プロセッサ1はdone1フラグを.true.に設定し、それを送った後実行を終了する。部分プロセッサ2は、フラグを受け取り、入力が終了したとみなして、実行を終了する。