第II部

High Performance Fortran 言語

ここでは、High Performance Fortran 言語 (バージョン 2.0) の機能の構文と意味について記述する。ここで使われているいくつかの技術用語は、第 I 部で定義されているか、さもなければ、その記述は内部に含まれている。また、第 III 部は、この資料に基づいている。

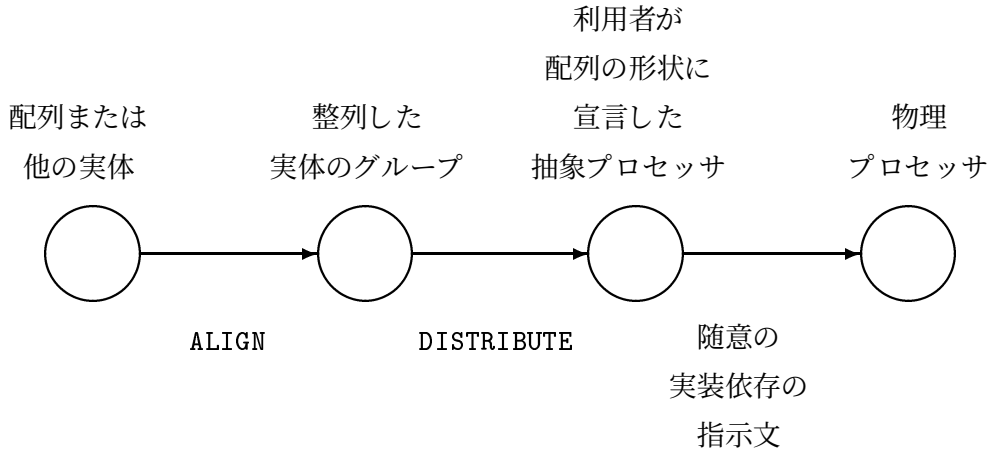
第3章　データマッピング

HPF のデータ整列とデータ分散の指示文は、どのように配列要素をプロセッサメモリへ割り当てるかを、プログラマがコンパイラに対して助言することを可能にする。本章では、利用可能なデータマッピング機能の中で基本的なもの、特に一つの有効域内で意味を持つ機能について論じる。第 4 章では、マップされた変数が手続の引数に現れた場合に適用される機能について論じる。

3.1　モデル

HPF は、いくつかの指示文を Fortran に追加している。それは、データ実体を複数のプロセッサメモリへ割り当てる方法を、利用者がコンパイラに対し助言するためのものである。このモデルではデータ実体からメモリ領域すなわち「抽象プロセッサ」へのマッピングに二つの段階が存在する。データ実体 (一般には配列要素) は、まず最初に、他のデータ実体に対して**整列**し、次に、この配列グループは抽象プロセッサの矩形構成へ**分散**される。(実装においては、これらの抽象プロセッサを実現するために同じ数かもっと少ない数の物理プロセッサが使われる。この抽象プロセッサの物理プロセッサへのマッピングは実装依存である。)

次の図は、このモデルを図示したものである。



ここには、二つの潜在的な仮定がある。一つは、2 個もしくはそれ以上のデータ実体に対する一つの演算は、それらのデータ実体すべてが同一のプロセッサに存在した場合に、より速く実行されるというものであり、もう一つは、そのような演算が多い場合は、個々の演算が別々のプロセッサで実行可能ならば、複数の演算が同時に実行される可能性があるというものである。

配列記述に代表される、多くの演算を同時に実行させるかどうかをコンパイラが決定しやすくする機能を、Fortran はいくつも提供している。HPF 指示文はデータ実体同士が同じ

プロセッサ上に存在すべきであるという推奨をコンパイラに知らせる手段を提供している: もし二つのデータ実体が同じ抽象プロセッサに (整列と分散の 2 段階のマッピングを経て) マップされていれば、それらは同じ物理プロセッサに存在するべきであるという、実装に対しての強力な推奨になる。また、一つのデータ実体を複数の場所に置くように推奨することもできる。これはデータ実体の更新処理を複雑にするかもしれないが、そのデータ実体を複数のプロセッサで読み込む場合の処理を速くする。

宣言文として機能する指示文と (Fortran 規格の意味における) 実行文として機能する指示文には明確な区別がある。宣言文はプログラム単位に入るときに実行され (それらはみな一度に同時に実行されるかのようであるが)、その後で実行文が実行される。(宣言文は翻訳時に処理されるものと考えると都合が良いことが多いが、それらのいくつかは、仮引数のような実行時に決まる量に依存することが許されている宣言式を含んでいる。したがって、それらの式は実行時、具体的にはプログラムの制御がその有効域に入った瞬間、にならないと利用できないかもしれない。)

基本的な概念として、すべての配列 (それどころか、すべてのデータ実体) は、ある言語要素への**何らか**の整列をもって生成される。そして、その言語要素は、抽象プロセッサの**何らか**の構成への**何らか**の分散をもつ。もし、配列 A の整列を他の配列 B に関連させて指定した明示的な宣言指示文がある場合には、A の分散は B の分散によって規定される。そうでない場合には、A の分散が明示的に示されているかも知れない。どちらの場合でも、このような明示的に宣言された情報は配列が生成されるときに使用される。

> 【実装者への助言】 このモデルは「配列はある暗黙の配置で作成され、明示的な指示文がある場合に再整列や再分散される」というようなモデルよりも、必要とされる仕事量においてより良い状況をもたらす。ALIGN と DISTRIBUTE という宣言指示文を使用しても、実装上の暗黙配置を用いるよりも、実行時においてより多くの仕事を発生させる必要はない。【以上】

割付け実体の場合は、実体は割り付けられるときに生成されると言える。割付け実体への宣言指示文は、プログラム単位の宣言部に現れるかも知れないが、効果をもつのは実体が生成されるときであり、制御が有効域に入るときではない。

整列はデータ実体の (Fortran の意味における) **属性**である。実体 A が、既に実体 C に整列している実体 B に整列している場合、A は C に直接に整列しているとみなされる。B は宣言時に単なる媒介の役割を果たすだけである。このとき、A は B には**直接的に整列している**と言い、A は C には**最終的に整列している**と言う。もしある実体が他の実体に明示的に整列していない場合には、自分自身に最終的に整列していると言う。整列関係は木構造を形作る。それは、根の実体に最終的に整列しているすべての実体から成る。しかし、その木はいつも直ちに「縮退」しているため、すべての実体は根へ直接の関係をもっている。

整列木の根であるすべての実体は、関連する**テンプレート**またはインデックス空間をもっている。通常、このテンプレートはそれに関連する実体と同じ次元数であり、各次元で同じ大きさをもっている。(この規則の最も重要な例外は INHERIT 属性をもった仮引数である。INHERIT については 4.4.2 項で解説されている。) 「配列のテンプレート」という言い回しは、よく使われる。これは配列が最終的に整列した実体のテンプレートを意味する。(明示的な TEMPLATE(3.7 節 参照) が使われているとき、単純にこのテンプレートが、配列が明示的に整列しているテンプレートになるかもしれない。)

HPF モデルの**分散**ステップは、テクニカルには配列のテンプレートに対して適用される。しかし、既に述べた密接な関係により、厳密ではないが、配列の分散として述べていることが多い。分散は、与えられたパターンに従って抽象プロセッサの集合に対しテンプレートを分割する。(配列からテンプレートへの) 整列と (テンプレートからプロセッサへの) 分散の連携は、したがって、配列のプロセッサへの関係を決定する。この関係を配列の**マッピング**と呼ぶ。(これは、スカラに対しても同様に適用される。それは、添字リストが空であることによって示される単一の点のみのインデックス空間をもつものとみなせるからである。)

すべての実体は、完全な宣言指示文のセットが存在するかのようにして生成される。もしプログラムがある実体のマッピングのための完全な宣言を含んでいない場合には、コンパイラは暗黙の宣言を提供する。暗黙の宣言によってある実体が他のいかなる実体とも整列しないとき、それは自分自身に最終的に整列している。暗黙の分散は実装依存であるが、明示的な指示文として表現可能でなくてはならない。宣言が同じ複数の実体に対して、同じ暗黙の分散宣言を提供する必要はない。例えば、コンパイラはその実体が実行コードで使われている文脈を考慮に入れてもよい。宣言が同じ実体同士が同じ分散をもつようにしたいならば、プログラマは暗黙に提供されるような分散を明示的に指定すればよい。(一方、宣言が同じプロセッサ構成同士は、「同じプロセッサ群を同じように構成する」 ことを表現することが保証されている。これについては 3.6 節で細かく論じる。)

ある場合には、一つの配列で全体のインデックス空間に渡る宣言は無いが、いくつかの小さな配列が整列している一つの大きなインデックス空間を考えることが望ましいことがある。HPF では、テンプレートを宣言することができる。テンプレートは分散することができ、配列を整列することができる単なる抽象インデックス空間である。これは、内容をもたない要素から成る配列のようなものであるため、記憶領域を使用しない。

実体は、その実体が宣言されている有効域にある HPF のマッピング指示文に現れた場合に、**明示的にマップされている**とみなされ、さもなければ**暗黙的にマップされている**とみなされる。マッピング指示文とは、`ALIGN`、`DISTRIBUTE`、`INHERIT` 指示文、それに、整列や分散または `INHERIT` 属性を与える指示文である。

実体の動的な再分散と再マッピングを許すために、第 8 章でこのモデルを拡張することに注意されたい。

3.2 データ整列とデータ分散指示文の構文

HPF における宣言指示文は二つの形式をもっている。Fortran の `DIMENSION` 文や `ALLOCATABLE` 文に似た文形式と、Fortran における “::” 区切りを使った型宣言文に似た属性形式である。

属性形式は、一つの指示文に一つ以上の属性を記述することを許している。HPF は最初の属性や、もちろんすべての属性に対して、型指定子でなければならないという制約がないことで、柔軟性において Fortran を超えている。

H301 *combined-directive* **is** *combined-attribute-list* :: *combined-decl-list*

H302 *combined-attribute* **is** ALIGN *align-attribute-stuff*
or DISTRIBUTE *dist-attribute-stuff*
or INHERIT
or TEMPLATE
or PROCESSORS
or DIMENSION (*explicit-shape-spec-list*)

H303 *combined-decl* **is** *hpf-entity* [(*explicit-shape-spec-list*)]
or *object-name*

H304 *hpf-entity* **is** *processors-name*
or *template-name*

INHERIT 属性は、サブルーチン呼出し規約に関連するものであり、第 4 章で論じる。

制約: 同じ種類の *combined-attribute* を、一つの *combined-directive* 中で 2 回以上指定してはならない。

制約: DIMENSION 属性が *combined-directive* に現れた場合、それが適用されるすべての言語要素は、HPF の TEMPLATE や PROCESSORS 型指定子で宣言されなければならない。

以下に述べる規約は、分かれた指示文にあるか *combined-directive* にあるかに関わらず、種々の属性の宣言を制約する。

DISTRIBUTE 属性がある場合は、 *combined-decl-list* に宣言されたすべての名前は、 *distributee* とみなされる。そして、それは 3.3 節で述べられている制約を受ける。

ALIGN 属性がある場合は、データ要素宣言並び (*entity-decl-list*) に宣言されたすべての名前は、*alignee* とみなされる。そして、それは 3.4 節で述べられている制約を受ける。

HPF のキーワードである PROCESSORS と TEMPLATE は、プロセッサ構成やテンプレートの宣言における型指定子の役割を果たす。HPF のキーワードである ALIGN、DISTRIBUTE および INHERIT は、属性の役割を果たす。プロセッサ構成、テンプレート、および他の型 (例えば REAL) の言語要素に関する属性の指定は、型指定子を伴わずに HPF 指示文と組み合わせることができる。

一つの言語要素には、ある属性を 2 回以上指定してはならない。

次元の情報は、*hpf-entity* の直後や、DIMENSION 属性の中に指定できる。両方が指定された場合には、実体名 (*object-name*) の直後に指定されたものが、DIMENSION 属性に指定されたものをくつがえす。(これは、 Fortran 規格での取り扱いと一致している。) 例えば、このようになる:

```
!HPF$ TEMPLATE,DIMENSION(64,64) :: A,B,C(32,32),D
```

A、B、および D は 64×64 のテンプレートである。C は 32×32 のテンプレートである。

変数をマップする指示文は、変数が宣言された有効域になければならない。

宣言式が同じ宣言部の中で宣言された配列要素の値を引用する場合、その配列に対する明示的なマッピングや INHERIT 属性は、先行する宣言指示文で完全に宣言されていなければならない。(この制約は、次に示す Fortran 規格の F95:7.1.6.2 により示唆され、また、それ

を拡張するものである: 宣言式が同じ宣言部の中で宣言された配列要素の値を引用する場合、その配列の上下限は、宣言部の先行する部分に宣言されていなければならない。)

アスタリスクに関する注釈: アスタリスク文字 "*" は、HPF の整列と分散指示文の構文規則では、別々の三つの役割をもつ。

- 丸括弧で囲まれた並び (list) のメンバである単一のアスタリスクは、縮退マッピングまたは複製マッピングを表す。縮退マッピングは、配列の多くの要素が一つの抽象プロセッサにマップされ、複製マッピングは、配列の一つの要素が多くの抽象プロセッサにマップされる。*align-source*、*align-subscript* (3.4 節 参照) および *dist-format* (3.3 節 参照) の構文規則を参照されたい。

- *align-subscript-use* 式に現れるアスタリスクは、通常の整数乗算演算子を表している。

- 左丸括弧 "(" の前やキーワード WITH や ONTO の後に現われるアスタリスクは、記述的または転写的マッピングを表わす。これらは、副プログラムの仮引数のマッピング (第 4 章 参照) や公認拡張仕様におけるポインタのマッピング (8.8 節 参照) のために用意されている。

アスタリスクは、 PASS_BY 属性でも使用される。これは、C 言語で書かれた外来ルーチンへ参照渡し される引数を記述する引用仕様宣言に現れる (11.2 節 参照)。

3.3 DISTRIBUTE 指示文

DISTRIBUTE 指示文は、プロセッサ構成で表された抽象プロセッサにデータ実体をマップすることを指定する。

```
      REAL SALAMI(10000)
!HPF$ DISTRIBUTE SALAMI(BLOCK)
```

これは、配列 SALAMI を、隣接する要素で構成されるブロックに均一にスライスし、ある抽象プロセッサの集合に分散することを指定している。50 台のプロセッサがあれば、それぞれが $\lceil 10000/50 \rceil = 200$ 個の要素をもつ複数のグループに分けられることを意味し、SALAMI(1:200) が最初のプロセッサにマップされ、次に SALAMI(201:400) が 2 番目のプロセッサにマップされ、さらに以下同じ ように続く。1 台のプロセッサしかなければ、10000 個の要素をもつ一つのブロックとして、配列のすべての要素がそのプロセッサにマップされる。

ブロックの大きさは、明示的に指定することもできる。

```
      REAL WEISSWURST(10000)
!HPF$ DISTRIBUTE WEISSWURST(BLOCK(256))
```

これは、256 個の要素からなるいくつかのグループが、連続する抽象プロセッサにマップされることを指定している。(この指示文が満たされるためには、少なくとも $\lceil 10000/256 \rceil = 40$ 個の抽象プロセッサが必要である。40 番目のプロセッサは、WEISSWURST(9985:10000) で示される 16 要素のみの不完全なブロックをもつことになる。

HPF はサイクリック分散形式も提供している:

```
      REAL DECK_OF_CARDS(52)
!HPF$ DISTRIBUTE DECK_OF_CARDS(CYCLIC)
```

4 台の抽象プロセッサがあるとすると、1 番目のプロセッサに DECK_OF_CARDS(1:49:4) があり、2 番目のプロセッサに DECK_OF_CARDS(2:50:4) があり、3 番目のプロセッサに DECK_OF_CARDS(3:51:4) があり、そして 4 番目のプロセッサに DECK_OF_CARDS(4:52:4) があることになる。連続した配列の要素はラウンドロビン風に連続した抽象プロセッサに分配される。

分散は、多次元配列の各次元にそれぞれ独立して指定される:

```
      INTEGER CHESS_BOARD(8,8), GO_BOARD(19,19)
!HPF$ DISTRIBUTE CHESS_BOARD(BLOCK, BLOCK)
!HPF$ DISTRIBUTE GO_BOARD(CYCLIC,*)
```

配列 CHESS_BOARD は、隣接する四角形の継ぎはぎに切り分けられ、2 次元に構成された抽象プロセッサに分散される。配列 GO_BOARD は、その行がサイクリックに、1 次元に構成された抽象プロセッサに分散される。("*" は GO_BOARD の 2 次元目が分割されていないことを指定している。したがって、完全な列が一つのオブジェクトとして分散されている。この形態は「オンプロセッサ」分散と呼ばれることがある。)

DISTRIBUTE 指示文は、有効域の宣言部にだけ現われることができる。また、BLOCK や CYCLIC オプションへの引数として、宣言式を含むことができる。

DISTRIBUTE 指示文の構文は次のとおりである。

H305	*distribute-directive*	**is**	DISTRIBUTE *distributee dist-directive-stuff*
H306	*dist-directive-stuff*	**is**	*dist-format-clause* [*dist-onto-clause*]
H307	*dist-attribute-stuff*	**is**	*dist-directive-stuff*
		or	*dist-onto-clause*
H308	*distributee*	**is**	*object-name*
		or	*template-name*
H309	*dist-format-clause*	**is**	(*dist-format-list*)
		or	* (*dist-format-list*)
		or	*
H310	*dist-format*	**is**	BLOCK [(*scalar-int-expr*)]
		or	CYCLIC [(*scalar-int-expr*)]
		or	*
H311	*dist-onto-clause*	**is**	ONTO *dist-target*
H312	*dist-target*	**is**	*processors-name*
		or	* *processors-name*
		or	*

完全を期すために、ここでは最大限の構文を示している。しかし、いくつかの形式は第 4 章でのみ論じられている。「手続境界」向けであるその形式は、次のものである。

- 規則 H309 の (*形式を含む) 下から二つのオプション。
- 規則 H312 の (*形式を含む) 下から二つのオプション。

制約: *distributee* である実体名 (*object-name*) は、単純な名前でなければならず、部分実体特定子や成分名 (*component-name*) であってはならない。

制約: *distributee* である実体名 (*object-name*) は、*alignee* であってはならない。

制約: *distributee* である実体名 (*object-name*) は、POINTER 属性をもつことはできない。

制約: *distributee* である実体名 (*object-name*) は、TARGET 属性をもつことはできない。

制約: *distributee* がスカラであるとき、*dist-format-list*(およびそれを囲む丸括弧) は現れてはならない。この場合、文形式の指示文は、*dist-format-clause* が "*" である場合だけが許される。

制約: *dist-format-list* が指定されたら、その長さはそれぞれの *distributee* の次元数と等しくなければならない。

制約: *dist-format-list* と *dist-target* の両方が指定されたとき、"*" でない *dist-format-list* の要素の数は、そのプロセッサ構成の次元数と等しくなければならない。

制約: *dist-format-list* がなく *dist-target* が指定されたとき、それぞれの *distributee* の次元数は、そのプロセッサ構成の次元数と等しくなければならない。

制約: DISTRIBUTE 指示文で、*dist-format-clause* または *dist-target* のどちらかが "*" で始まるときは、すべての *distributee* は仮引数でなければならない。

制約: DISTRIBUTE 指示文の *dist-format* 中の整数式 (*scalar-int-expr*) は、すべて宣言式 (*specification-expr*) でなくてはならない。

【利用者への助言】 上記の制約のいくつかは、公認拡張仕様で緩和される。(第 8 章 参照): 構造型成分のマッピング (制約 1 を緩和する)、それに、ポインタや指示先のマッピング (制約 3、4 および 9 を緩和する) である。【以上】 ⇓

次の形式の DISTRIBUTE 指示文が、

!HPF$ DISTRIBUTE *dist-attribute-stuff* :: *distributee-list*

combined-directive 向けの構文規則 H301 によって可能になっていることに注意されたい。

例:

```
!HPF$ DISTRIBUTE D1(BLOCK)
!HPF$ DISTRIBUTE (BLOCK,*,BLOCK) ONTO SQUARE:: D2,D3,D4
```

別の面から見た *dist-format* の意味は次のとおりである。

切上げ除算関数を `CD(J,K) = (J+K-1)/K` として定義する。(Fortran の整数計算におけるゼロ方向への切捨てを用いている。)

切上げ剰余関数を `CR(J,K) = J-K*CD(J,K)` として定義する。

dist-target として現れているプロセッサ構成の各次元は、左から右の順で、 *dist-format* が `*` ではない *distributee* の次元に**対応している**といえる。上記の例では、プロセッサ構成 `SQUARE` は 2 次元でなければならず、1 次元目は `D2`、`D3` および `D4` の 1 次元目に対応していて、2 次元目は `D2`、`D3` および `D4` の 3 次元目に対応している。

distributee のある次元における大きさを d とし、プロセッサ構成のそれに対応する次元における大きさを p としよう。単純にするために、すべての次元における下限は 1 とする。そうすると、 `BLOCK(`m`)` の意味は次のようになる。その次元における、添字が j の *distributee* の位置は、プロセッサ構成の対応する次元に沿った `CD(`j`,`m`)` なる添字をもつ抽象プロセッサにマップされ ($m \times p \geq d$ が真でなくてはならないことに注意)、マップされた抽象プロセッサの中での位置は、m`+CR(`j`,`m`)` である。抽象プロセッサ k のその軸に沿った最初の *distributee* の位置は、`1+`m`*(`k`-1)` である。

ブロックの大きさ m は、正の整数でなければならない。

定義によって `BLOCK` は `BLOCK(CD(`d`,`p`))` と同じ意味をもつ。

`CYCLIC(`m`)` の意味は、次のようになる。その次元における、添字が j の *distributee* の位置は、プロセッサ構成の対応する次元に沿った `1+MODULO(CD(`j`,`m`)-1,`p`)` なる添字をもつ抽象プロセッサにマップされる。抽象プロセッサ k のその軸に沿った最初の *distributee* の位置は、`1+`m`*(`k`-1)` である。

ブロックの大きさ m は、正の整数でなければならない。

定義によって `CYCLIC` は `CYCLIC(1)` と同じ意味をもつ。

`CYCLIC(`m`)` と `BLOCK(`m`)` は、 $m \times p \geq d$ である場合に、同じ分散を意味するが、`BLOCK(`m`)` はこれに加えて、分散がサイクリックに回らないことを表明している。このことは m が定数でないときには、コンパイラには断言することができない。`CYCLIC` と (引数の式をもたない)`BLOCK` は、 $p \geq d$ 、つまりブロックの大きさが 1 で、分散が循環的に回らないという変則的な場合でなければ、同じ分散を意味しないことに注意されたい。

16 台の抽象プロセッサと長さ 100 の配列を考える。

```
!HPF$ PROCESSORS SEDECIM(16)
      REAL CENTURY(100)
```

配列を `BLOCK` (この場合には `BLOCK(7)` と同じことを意味する) に分散することによって、

```
!HPF$ DISTRIBUTE CENTURY(BLOCK) ONTO SEDECIM
```

抽象プロセッサ上に、次のように配列要素をマップすることになる。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	8	15	22	29	36	43	50	57	64	71	78	85	92	99	
2	2	9	16	23	30	37	44	51	58	65	72	79	86	93	100	
3	3	10	17	24	31	38	45	52	59	66	73	80	87	94		
4	4	11	18	25	32	39	46	53	60	67	74	81	88	95		
5	5	12	19	26	33	40	47	54	61	68	75	82	89	96		
6	6	13	20	27	34	41	48	55	62	69	76	83	90	97		
7	7	14	21	28	35	42	49	56	63	70	77	84	91	98		

BLOCK(8) に配列を分散することによって、

```
!HPF$ DISTRIBUTE CENTURY(BLOCK(8)) ONTO SEDECIM
```

抽象プロセッサ上に、次のように配列要素をマップすることになる。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	9	17	25	33	41	49	57	65	73	81	89	97			
2	2	10	18	26	34	42	50	58	66	74	82	90	98			
3	3	11	19	27	35	43	51	59	67	75	83	91	99			
4	4	12	20	28	36	44	52	60	68	76	84	92	100			
5	5	13	21	29	37	45	53	61	69	77	85	93				
6	6	14	22	30	38	46	54	62	70	78	86	94				
7	7	15	23	31	39	47	55	63	71	79	87	95				
8	8	16	24	32	40	48	56	64	72	80	88	96				

BLOCK(6) に配列を分散することは、$6 \times 16 < 100$ であることから、HPF 規格合致ではない。

配列を CYCLIC (常に CYCLIC(1) と同じことを意味する) に分散することによって、

```
!HPF$ DISTRIBUTE CENTURY(CYCLIC) ONTO SEDECIM
```

抽象プロセッサ上に、次のように配列要素をマップすることになる。

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
65	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
81	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
97	97	98	99	100												

CYCLIC(3) に配列を分散することによって、

```
!HPF$ DISTRIBUTE CENTURY(CYCLIC(3)) ONTO SEDECIM
```

抽象プロセッサ上に、次のように配列要素をマップすることになる。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	4	7	10	13	16	19	22	25	28	31	34	37	40	43	46
2	5	8	11	14	17	20	23	26	29	32	35	38	41	44	47
3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48
49	52	55	58	61	64	67	70	73	76	79	82	85	88	91	94
50	53	56	59	62	65	68	71	74	77	80	83	86	89	92	95
51	54	57	60	63	66	69	72	75	78	81	84	87	90	93	96
97	100														
98															
99															

いくつかのプロセッサが一つの要素ももたないように配列を分散することが、完全に許されていることに注意されたい。それどころか、配列のすべての要素が一つのプロセッサに、「分散される」ことも許されている。例えば、次の例は、

```
!HPF$ DISTRIBUTE CENTURY(BLOCK(256)) ONTO SEDECIM
```

ただ一つの空でない (100 要素だけ部分的に詰められた) ブロックが、1 番目のプロセッサに割り当てられ、2 番目から 16 番目のプロセッサには一つの要素も割り当てられない。

DISTRIBUTE 指示文の文形式は、一つだけの *distributee* を指定しようとした属性形式の短縮形と考えることができる。

```
!HPF$ DISTRIBUTE distributee ( dist-format-list ) ONTO dist-target
```

は、以下と同じである。

```
!HPF$ DISTRIBUTE ( dist-format-list ) ONTO dist-target :: distributee
```

構文的な曖昧さを防ぐために、文形式では *dist-format-clause* が必要である。そのため、一般にはスカラのマッピングを指定するために文形式の指示文が使われることはないことに注意されたい。

属性形式において *dist-format-clause* が省略されていれば、処理系はそれぞれのテンプレートや配列ごとに、任意の分散を選択することができる。したがって、次の指示文は、

```
!HPF$ DISTRIBUTE ONTO P :: D1,D2,D3
```

以下の指示文と同じであり、

```
!HPF$ DISTRIBUTE ONTO P :: D1
!HPF$ DISTRIBUTE ONTO P :: D2
!HPF$ DISTRIBUTE ONTO P :: D3
```

コンパイラは、プログラム中で D1、D2 および D3 が使用されるパターンを考慮した上で、次のような三つの別々の分散を与えることを選択するかも知れない。

```
!HPF$ DISTRIBUTE D1(BLOCK, BLOCK) ONTO P
!HPF$ DISTRIBUTE D2(CYCLIC, BLOCK) ONTO P
!HPF$ DISTRIBUTE D3(BLOCK(43),CYCLIC) ONTO P
```

また、これらの三つの配列に対して、たまたま一つの同じ分散を選択してしまうこともあるかもしれない。

文形式または属性形式のどちらでも、ONTO 節があれば、それは分散先であるプロセッサ構成を指定している。ONTO 節が省略されていれば、それぞれの *distributee* ごとに、実装依存のプロセッサ構成が任意に選択される。したがって、例えば次の例では、

```
      REAL, DIMENSION(1000) :: ARTHUR, ARNOLD, LINUS, LUCY
!HPF$ PROCESSORS EXCALIBUR(32)
!HPF$ DISTRIBUTE (BLOCK) ONTO EXCALIBUR :: ARTHUR, ARNOLD
!HPF$ DISTRIBUTE (BLOCK) :: LINUS, LUCY
```

配列 ARTHUR と ARNOLD は同じマッピングとなり、対応する要素同士は同じ抽象プロセッサに存在する。なぜなら、それらは同じ大きさの配列であり、同じプロセッサ構成 (EXCALIBUR) に対して同じよう (BLOCK) に分散されたからである。しかし、配列 LUCY と LINUS は、必ずしも同じマッピングであるとは限らない。なぜなら、それらは、実装依存で、別々に選択されたプロセッサ構成に対して分散されるからである。LUCY と LINUS の対応する要素同士は、同じ抽象プロセッサには存在しないかも知れない。(ALIGN 指示文は、プロセッサ構成を明示的に指定しなくても、二つの配列が同じマッピングであることを保証する方法を提供している。)

与えられた環境の中では、ある分散に対しては、適切なプロセッサ構成が存在しないかもしれない。

3.4 ALIGN 指示文

ALIGN 指示文は、あるデータ実体が他のあるデータ実体と同じようにマップされることを指定するために使用される。整列したデータ実体間の演算は、整列しているかどうか判らないデータ実体間の演算よりも、おそらく効率が良いはずである。(なぜなら、整列している二つの実体は、コンパイラが同一の抽象プロセッサにマップしようとするからである。) ALIGN 指示文は、特に配列のすべての要素に関する明示的なマッピングを一度に指定することが簡単にできるように設計されている。ある場合には、揃いの DISTRIBUTE 指示文を注意して使うことで、複数の実体を整列させることができるが、ALIGN はより一般的であり、より便利なことが多い。

ALIGN 指示文は、有効域の宣言部にだけ現われることができる。また、添字 (*subscript*) や 添字三つ組 (*subscript-triplet*) として、宣言式を含むことができる。

ALIGN 指示文の構文は次のとおりである。

H313 *align-directive* is ALIGN *alignee* *align-directive-stuff*

H314 *align-directive-stuff* **is** (*align-source-list*) *align-with-clause*

H315 *align-attribute-stuff* **is** [(*align-source-list*)] *align-with-clause*

H316 *alignee* **is** *object-name*

H317 *align-source* **is** :
or *
or *align-dummy*

H318 *align-dummy* **is** *scalar-int-variable*

制約: *alignee* である実体名 (*object-name*) は、単純な名前でなければならず、部分実体特定子や成分名 (*component-name*) であってはならない。

制約: *alignee* である実体名 (*object-name*) は、*distributee* であってはならない。

制約: *alignee* である実体名 (*object-name*) は、`POINTER` 属性をもつことはできない。

制約: *alignee* である実体名 (*object-name*) は、`TARGET` 属性をもつことはできない。

制約: *alignee* がスカラであるとき、*align-source-list* (およびそれを囲む丸括弧) は現れてはならない。この場合、文形式の指示文は許されない。

制約: *align-source-list* が指定されたとき、その長さは *alignee* の次元数と等しくなければならない。

制約: *align-dummy* は、名前付き変数でなければならない。

制約: 実体は、 `INHERIT` 属性と `ALIGN` 属性の両方をもつことはできない。

⇓

【利用者への助言】 上記の制約のいくつかは、公認拡張仕様で緩和される。(第 8 章 参照): 構造型成分のマッピング (制約 1 を緩和する)、それに、ポインタや指示先のマッピング (制約 3 と 4 を緩和する) である。【以上】

次の形式の `ALIGN` 指示文が、

`!HPF$ ALIGN` *align-attribute-stuff* `::` *alignee-list*

combined-directive 向けの構文規則 H301 によって可能になっていることに注意されたい。

`ALIGN` 指示文の文形式は、一つだけの *alignee* を指定しようとした属性形式の短縮形と考えることができる。

`!HPF$ ALIGN` *alignee* `(` *align-source-list* `) WITH` *align-spec*

は、以下と同じである。

`!HPF$ ALIGN (` *align-source-list* `) WITH` *align-spec* `::` *alignee*

属性形式で、*align-source-list* が省略されていて、*alignee* がスカラでないとき、*align-source-list* は、*alignee* の次元数と同じ数の “:” の並びを括弧で囲んだものであるとみなされる。同様に、文形式および属性形式で、*align-spec* の *align-subscript-list* が省略されているとき、それは *align-target* の次元数と同じ数の “:” の並びを括弧で囲んだものであるとみなされる。したがって、次の指示文は、

```
!HPF$ ALIGN WITH B :: A1, A2, A3
```

以下を意味する。

```
!HPF$ ALIGN (:,:) WITH B(:,:) :: A1, A2, A3
```

これはさらに、次の指示文と同じである。

```
!HPF$ ALIGN A1(:,:) WITH B(:,:)
!HPF$ ALIGN A2(:,:) WITH B(:,:)
!HPF$ ALIGN A3(:,:) WITH B(:,:)
```

なぜなら、一つ以上の *alignee* を指定した属性形式指示文は、一つの指示文に一つの *alignee* だけを指定した同じような指示文の羅列と等しいからである。ここでは、すべての *alignee* は同じ次元数でなければならない。このような理解のもとで、以下での記述を単純にするために、ALIGN 指示文は一つだけの *alignee* をもつと仮定しよう。

それぞれの *align-source* は *alignee* の一つの軸に対応し、そこには “:”、“*” またはダミー変数が指定されている。

- “:” であれば、その軸上のすべての点は、*align-spec* の対応する軸上にばらまかれる (以下を参照)。
- “*” であれば、その軸は **縮退** している。すなわち、その軸上の点は *align-target* の対応する点を決める場合に無関係である。(“*” を、その指示文の他のどこにも使われていないダミー変数名と置き換えても同じ効果が得られる。“*” は、変数名を用意する手間を省き、その次元には何の依存も意図されていないことを明確にするために便利であるので用意されている。)
- ダミー変数は、*alignee* のその次元で有効なすべてのインデックス値の間で変動するとみなされる。

ALIGN 指示文の WITH 節の構文は次のとおりである。

H319	*align-with-clause*	**is** WITH *align-spec*
H320	*align-spec*	**is** *align-target* [(*align-subscript-list*)] **or** * *align-target* [(*align-subscript-list*)]
H321	*align-target*	**is** *object-name* **or** *template-name*

H322 *align-subscript* **is** *int-expr*
or *align-subscript-use*
or *subscript-triplet*
or *****

H323 *align-subscript-use* **is** [[*int-level-two-expr*] *add-op*]
align-add-operand
or *align-subscript-use add-op int-add-operand*

H324 *align-add-operand* **is** [*int-add-operand* *****] *align-primary*
or *align-add-operand* ***** *int-mult-operand*

H325 *align-primary* **is** *align-dummy*
or **(** *align-subscript-use* **)**

H326 *int-add-operand* **is** *add-operand*

H327 *int-mult-operand* **is** *mult-operand*

H328 *int-level-two-expr* **is** *level*-2-*expr*

完全を期すために、ここでは最大限の構文を示している。しかし、いくつかの形式は第 4 章 でのみ論じられている。「手続境界」向けであるその形式は、規則 H320 の (*形式を含む) 第二のオプションである。

制約: *align-target* である実体名 (*object-name*) は、単純な名前でなければならず、部分実体特定子や成分名 (*component-name*) であってはならない。

制約: *align-target* は、 `OPTIONAL` 属性をもつことはできない。

制約: `ALIGN` 指示文の *align-spec* が "*" で始まるときは、すべての *alignee* は仮引数でなければならない。

制約: *align-directive* 中の整数式 (*int-expr*)、*int-level-two-expr*、*int-add-operand* および *int-mult-operand* は、宣言式でなくてはならない。

制約: *align-directive* 中の *align-subscript* である添字三つ組 (*subscript-triplet*) の添字 (*subscript*) や刻み幅 (*stride*) は、宣言式でなくてはならない。

制約: *align-subscript-list* では、同一の *align-dummy* は高々1 回だけ現れることができる。

制約: *align-subscript-use* 式には、*align-dummy* は高々1 回だけ現れることができる。

制約: *align-dummy* として使われた *scalar-int-variable* は、上記の文法により明確に許された場所以外では *align-spec* の中には現れてはならない。言い代えれば、一つの *align-dummy* に、*align-dummy* を含まない整数宣言式を加数や乗数として加えることでのみ、*align-subscript-use* を構成することができる。

制約: *align-subscript* 中の添字 (*subscript*) には、いかなる *align-dummy* も現れてはならない。

制約: *int-add-operand*, *int-mult-operand* および *int-level-two-expr* は、整数型でなくてはならない。

【利用者への助言】 上記の制約のいくつかは、公認拡張仕様で緩和される。(第 8 章 参照): 構造型成分のマッピング (制約 1 を緩和する)、ポインタのマッピング (制約 3 を緩和する)、それに、データ実体の再マッピング (制約 4 と 5 を緩和する) である。【以上】 ⇓

align-subscript-use に関する構文規則は、演算子の優先順位問題を考慮しているため込み入っているが、基本的な考え方は単純である: *align-subscript-use* は、一つだけ現れる *align-dummy* の線形関数 (もっと正確に言うとアフィン関数) になるように意図されている。

例えば、次に示す *align-subscript-use* 式は、みな正しい。ここで、 J、 K および M は、*align-dummy* であり、N は *align-dummy* でないとする:

```
J     J+1       3-K       2*M      N*M         100-3*M
-J    +J        -K+3      M+2**3   M+N         -(4*7+IOR(6,9))*K-(13-5/3)
M*2   N*(M-N)   2*(J+1)   5-K+3    10000-M*3   2*(3*(K-1)+13)-100
```

次に示す *align-subscript-use* 式は、みな正しくない。

```
J+J   J-J         3*K-2*K   M*(N-M)   2*J-3*J+J   2*(3*(K-1)+13)-K
J*J   J+K         3/K       2**M      M*K         K-3*M
K-J   IOR(J,1)    -K/3      M*(2+M)   M*(M-N)     2**(2*J-3*J+J)
```

align-spec は、*align-source-list* に現れているコロン (":") の数と同じ数の添字三つ組 (*subscript-triplet*) を含んでいなければならない。これらは、コロンでない *align-source* や添字三つ組 (*subscript-triplet*) でない *align-subscript* を除いて、左から右の順で対応付けられる。*align-source* がコロンとなっている *alignee* の次元を考えてみる。ここで、その次元の下限と上限をそれぞれ LA、UA とし、対応する *align-target* の添字三つ組は、LT:UT:ST であるとする。このとき、コロンはまだ使われていない新たなダミー変数 (ここでは J とする) で置き換えることが可能であり、添字三つ組は、指示文で指定されたマッピングを変えることなく (J-LA)*ST+LT という式で置き換えることができる。ただし、コロン形式は、その軸が

$$\max(0, UA - LA + 1) \;=\; \max(0, \lceil (UT - LT + 1)/ST \rceil)$$

が真であることを満たさなければならない。(これは、配列代入文の扱いとよく似ている。)

残りの議論を簡単にするために、 *align-source-list* 中のすべてのコロンは、上記の方法で新しいダミー変数で置き換えられているとする。同様に、 *align-source-list* 中のすべての "*" は、使われていない別のダミー変数で置き換えられているとする。例えば、

```
!HPF$ ALIGN A(:,*,K,:,:,*) WITH B(31:,:,K+3,20:100:3)
```

は、次のように変換できる。

```
!HPF$ ALIGN A(I,J,K,L,M,N) WITH B(I-LBOUND(A,1)+31,       &
!HPF$            L-LBOUND(A,4)+LBOUND(B,2),K+3,(M-LBOUND(A,5))*3+20)
```

必要条件は次のとおりである。

SIZE(A,1) .EQ. UBOUND(B,1)-30
SIZE(A,4) .EQ. SIZE(B,2)
SIZE(A,5) .EQ. (100-20+3)/3

したがって、以降では、すべての *align-source* がダミー変数であり、*align-subscript* が添字三つ組 (*subscript-triplet*) ではない場合についてだけ検討すればよい。

ダミー変数は、 *alignee* の対応する次元で有効なすべてのインデックス値の間で変動するとみなされる。各次元ごとのインデックス変数の取り得る値を組み合わせたものが、*alignee* の一つの要素に対応する。*align-spec* は、*alignee* の要素が整列する *align-target* の要素 (または部分) を指示している。この指示はインデックス値の関数であり得るが、その性質は線形関数 (もっと正確に言うとアフィン関数) であると (先に述べたように) 構文上で制約されている。これは、実装の複雑さを限定するためである。*align-dummy* 変数は、*align-spec* に高々1回だけ、厳密に規定された文脈においてだけ現れてもよい。その結果、*align-subscript* 式は、高々一つの *align-dummy* 変数を含むことができ、その変数の一次関数でなければならないと強制される。(したがって、斜めの整列は不可能である。)

align-subscript がアスタリスク "*" のときは、複製を表現している。*alignee* の要素はそれぞれ、*align-target* の軸上のすべての点に整列する。

【仕様の根拠】 縮退と複製の意味の両方で "*" を使うのは奇異に思えるかも知れないが、その原理はこうである。"*" は、概念的には常に、その式のどこにも現れていないダミー変数を意味する。そしてその変数は、指定された次元のインデックス集合の中で変動する。例えば、次の例は、

```
!HPF$ ALIGN A(:) WITH D(:,*)
```

A のコピーが一つずつ、D の列に整列している。なぜなら、それは概念的には以下と同等であるからである。

すべての正当なインデックス j において、 *align* A(:) *with* D(:,j)

同様に、次の例は、

```
!HPF$ ALIGN A(:,*) WITH D(:)
```

概念的には以下と同等である。

すべての正当なインデックス j において、 *align* A(:,j) *with* D(:)

HPF の構文では、次の文を、

```
!HPF$ ALIGN A(:,*) WITH D(:)
```

次のように書き換えることは、許されているが、

```
!HPF$ ALIGN A(:,J) WITH D(:)
```

次の文を、

```
!HPF$ ALIGN A(:) WITH D(:,*)
```

次のように書き換えることは、許されていないことに注意されたい。

```
!HPF$ ALIGN A(:) WITH D(:,J)
```

なぜなら、それは別の意味をもつからである。(*alignee* に続く *align-source-list* に現れた変数だけが、*align-dummy* であると解釈される。そのためこの例では、変数 J の現在の値が使われ、A は D の一つの列に整列する。)

複製は、最適化コンパイラが最も近くにあるコピーを読むようなコードを生成することを可能にする。(もちろん、複製されたデータ実体が書き換えられた時には、一つのコピーだけでなく、すべてのコピーを変更しなければならない。複製表現は、小さなルックアップテーブルにとって大変有用である。アルゴリズムにとって論理的には不必要な余分な次元を与えることなしに、物理プロセッサごとにコピーをもつことで、大幅に速くなるからである。【以上】

上記の変換を適用することによって、*align-subscript* のすべての場合は、概念的には整数式 (*int-expr*)(*align-dummy* を伴わない) または *align-subscript-use* のどちらかに還元することができる。*align-source-list* は、“*” や “:” を含まないインデックス変数からなるリストに還元することができる。ここで *align-subscript-list* は、*align-dummy* 変数が取り得る値の組合せごとに *align-subscript* を単に式として評価することで、評価される。結果として得られる添字の値は、*align-target* にとって正当な添字でなければならない。(これは、*alignee* は、*align-target* の「端を超えて循環する」ことや、「端を超えて伸びる」ことが許されていないことを意味している。) そして、*alignee* の選択された要素は、こうして示された *align-target* の要素に整列すると考えられる。もっと正確に言うと、*alignee* の選択された要素は、示された *align-target* の要素がそのとき最終的に整列している実体 (自分自身であることもある) と、最終的に整列していると考えられる。

ALIGN 指示文の例をさらに示す:

```
      INTEGER D1(N)
      LOGICAL D2(N,N)
      REAL, DIMENSION(N,N):: X,A,B,C,AR1,AR2A,P,Q,R,S
!HPF$ ALIGN X(:,*) WITH D1(:)
!HPF$ ALIGN (:,*) WITH D1:: A,B,C,AR1,AR2A
!HPF$ ALIGN WITH D2:: P,Q,R,S
```

alignee-list では、すべての *alignee* は同じ次元数でなければならないが、形状は同じである必要はないことに注意されたい。各次元の寸法は、*align-source-list* でコロンに該当する次元で一致していなければならないだけである。このことは便宜上で大変重要なことである。実際によくある状況は、分散された (「並列」) 次元では大きさが一致するが、縮退された (「オンプロセッサ」) 次元では大きさが一致しない複数の配列を、一度に整列する場合である。

```
      REAL A(3,N), B(4,N), C(43,N), Q(N)
!HPF$ DISTRIBUTE Q(BLOCK)
!HPF$ ALIGN (*,:) WITH Q:: A,B,C
```

この例では、プロセッサ (おそらく N 台) と、各プロセッサでの大きさ (それぞれ 3, 4, 43) の違う三つの配列が要求されている。HPF に関する限り、それらの大きさが 3, 4 それに 43 と違っていても構わない。なぜなら、その軸は縮退しているからである。したがって、その軸だけに沿って変化するインデックスが指す配列要素は、すべて Q の同じ要素に整列される。(それは結局、同じプロセッサに存在するように指定されていることになる。)

次にあげる例では、同じグループの指示文は、みな同じ意味を表している。ただし、対応する軸の上限と下限は一致していると仮定している。

```
! X の二つ目の軸は縮退している。
!HPF$ ALIGN X(:,*) WITH D1(:)
!HPF$ ALIGN X(J,*) WITH D1(J)
!HPF$ ALIGN X(J,K) WITH D1(J)

! D3 の二つ目の軸は複製表現である。
!HPF$ ALIGN X(:,:) WITH D3(:,*,:)
!HPF$ ALIGN X(J,K) WITH D3(J,*,K)

! 二つの軸を入れ換えている。
!HPF$ ALIGN X(J,K) WITH D2(K,J)
!HPF$ ALIGN X(J,:) WITH D2(:,J)
!HPF$ ALIGN X(:,K) WITH D2(K,:)
! 両方のインデックス変数を除く方法はない。
! subscript-triplet 構文だけでは入れ換えを表現できないためである。

! 両方の軸で順序を逆にしている。
!HPF$ ALIGN X(J,K) WITH D2(M-J+1,N-K+1)
!HPF$ ALIGN X(:,:) WITH D2(M:1:-1,N:1:-1)

! 簡単な例
!HPF$ ALIGN X(J,K) WITH D2(J,K)
!HPF$ ALIGN X(:,:) WITH D2(:,:)
!HPF$ ALIGN (J,K) WITH D2(J,K):: X
!HPF$ ALIGN (:,:) WITH D2(:,:):: X
!HPF$ ALIGN WITH D2:: X
```

3.5 割付け配列とポインタ

ALIGN 指示文の *alignee* または DISTRIBUTE 指示文の *distributee* として、ALLOCATABLE 属性をもつ変数が現れることがある。これらの指示文は、有効域の入口で意味をもつのではなく、

ALLOCATE 文で配列が割り付けられるときに効果がある。これらの指示文の中にあるすべての宣言式の値は有効域の入口で一度だけ評価されるが、何度でも使用されてよい (使用されなくてもよい)。以下に例を示す。

```
      SUBROUTINE MILLARD_FILLMORE(N,M)
      REAL, ALLOCATABLE, DIMENSION(:) :: A, B
!HPF$ ALIGN B(I) WITH A(I+N)
!HPF$ DISTRIBUTE A(BLOCK(M*2))
      N = 43
      M = 91
      ALLOCATE(A(27))
      ALLOCATE(B(13))
      ...
```

副プログラム入口の式 N と M*2 の値は、概念的には ALIGN 指示文と DISTRIBUTE 指示文が保持していて、後の割付け時に使用される。配列 A は割り付けられるときに分散されるが、そのブロックの大きさは 182 ではなく、保持されている M*2 の値となる。配列 B は割り付けられるときに A に対して整列するが、そのときの N の値は 43 ではなく、保持されている値となる。

MILLARD_FILLMORE の例で、二つの ALLOCATE 文の順序を入れ替えると、正しくないプログラムになることに注意されたい。一般に、データ実体 X が生成されるときに別のデータ実体 Y に対して整列する場合には、Y は既に生成されているか割り付けられていなければならない。関連する例を以下に示す。

```
      SUBROUTINE WARREN_HARDING(P,Q)
      REAL P(:)
      REAL Q(:)
      REAL R(SIZE(Q))
      REAL, ALLOCATABLE :: S(:),T(:)
!HPF$ ALIGN P(I) WITH T(I)                    ! 規格合致でない
!HPF$ ALIGN Q(I) WITH *T(I)                   ! 規格合致でない
!HPF$ ALIGN R(I) WITH T(I)                    ! 規格合致でない
!HPF$ ALIGN S(I) WITH T(I)
      ALLOCATE(S(SIZE(Q)))                    ! 規格合致でない
      ALLOCATE(T(SIZE(Q)))
```

三つの ALIGN 指示文は、整列が行われるときに配列 T が割り付けられていないため、HPF 規格合致ではない。この四つの ALIGN 指示文は、それぞれ少しずつ状況が違う。配列 P と Q はサブルーチンの入口で既に存在しているが、T はまだ割り付けられていない。そのため、P を T に対して正しく指令的に整列することはできず、また、同様に Q を記述的に整列することはできない。(指令的と記述的に関する議論は、第 4 章を参照されたい。) 配列 R はサブルーチン入口で生成され、その大きさは Q の SIZE に依存するが、R の整列は T の整列に依存してはならない。これは、R の大きさが T の大きさに依存してはならないのと同じである。S の整列は S が割り付けられるまで起こらないので、S を T に対して整列する指示文は**書いてよい**。し

かし、Sが整列されるときにTはまだ割り付けられていないので、最初のALLOCATE文はHPF規格合致ではない。

割付け配列に対して明示的なALIGN指示文がある場合、配列の割付け時の整列は既に存在している実体またはテンプレートに対して行われる。明示的なALIGN指示文がない場合、配列は自分自身に対して最終的に整列する。配列が解放によって不定となるときに、他のデータ実体がその配列に対して最終的に整列していてはならない。この制約は、配列を生成するときにALLOCATE文で使われた名前がALLOCATABLE属性をもつかPOINTER属性をもつかに関わらず適用される。

HPFではポインタは明示的にマップできないので、明示的にマップされないデータ実体とだけ結合できる。ポインタを割付けに使用するとき、コンパイラはポインタを通して割り付けられるデータに任意のマッピングを選んでよい。ポインタの明示的なマッピングは公認拡張仕様の中では許されている (詳細は8.8節を参照されたい)。また、ポインタとSEQUENCE属性の関係については3.8節で記述する。

3.6 PROCESSORS指示文

PROCESSORS指示文は、一つまたは複数の矩形状のプロセッサ構成を宣言し、それぞれの名前と次元数と各次元の寸法を指定する。この指示文は、有効域の宣言部にだけ現れることができる。プロセッサ構成はどの次元も寸法が0であってはならない。したがって、プロセッサ構成は空になることはない。

F95:14.1.2の用語によれば、プロセッサ構成は類(1)の局所要素である。したがって、プロセッサ構成は同じ有効域の中で、変数や名前付き定数や内部手続などと同じ名前であってはならない。プロセッサ構成の名前は、F95:12.1.2.2.1のリストの中の他の名前と同じように親子結合と参照結合の規則に従う。

モジュール内で宣言されたプロセッサ構成は、そのモジュールの暗黙の参照許可属性をもつ。

> 【仕様の根拠】プロセッサ構成の名前は、HPFの第一種言語要素ではなく指示文の中にだけ現れなければならないため、参照許可宣言文(PRIVATEとPUBLIC)に現れることができない。もしも指示文が構造化された注釈ではなく通常のFortranの文であるなら、プロセッサ構成の参照許可は参照許可宣言文に名前を指定することによって制御できるようにするのがよいであろう。【以上】

二つのプロセッサ構成が同じ形状のとき、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と解釈される。(いくつかのHPFの実装で提供される実装依存の指示文では、同一の形状をもつプロセッサ構成の暗黙の関係を無効にできるようである。)

プログラム実行中のある時点で、指示文の指定によって二つのデータ実体を同じ抽象プロセッサにマップさせることは、同じ物理プロセッサにマップさせることを意味する。

組込み関数のNUMBER_OF_PROCESSORSとPROCESSORS_SHAPEは、実際にプログラムを実行する物理プロセッサの総数の問合せのために使用することができる。この情報は、宣言される抽象プロセッサ構成の適切な大きさを計算するために使うことができる。

H329 *processors-directive* **is** PROCESSORS *processors-decl-list*

H330 *processors-decl* **is** *processors-name*
[(*explicit-shape-spec-list*)]

例:

```
!HPF$ PROCESSORS P(N)
!HPF$ PROCESSORS Q(NUMBER_OF_PROCESSORS()),       &
!HPF$            R(8,NUMBER_OF_PROCESSORS()/8)
!HPF$ PROCESSORS BIZARRO(1972:1997,-20:17)
!HPF$ PROCESSORS SCALARPROC
```

形状が指定されていない場合、宣言されたプロセッサ構成はスカラであると考える。

> 【仕様の根拠】スカラプロセッサ構成が有効となるのは、スカラデータについて、一ヵ所にまとめて保持する必要はあるが分散データとの間には強い相互依存関係はないということを指示したい場合であろう。このようなプロセッサ構成に分散されたデータは、実装するアーキテクチャによって、単一の「コントロール」プロセッサや「ホスト」プロセッサに置かれることもあり (もしマシンがそういうものをもつなら)、任意に選ばれた1台のプロセッサに置かれることもあり、すべてのプロセッサに複製されることもある。複数の計算プロセッサと別個のスカラホスト計算機から成るようなターゲットアーキテクチャに対しては、自然な実装はすべてのスカラプロセッサ構成をホストプロセッサにマップすることである。複数の計算プロセッサから成り別個のスカラ「ホスト」計算機をもたないなら、プロセッサ構成はある任意に選ばれた一つの計算プロセッサにマップされるかもしれないし、すべての計算プロセッサに複製されるかもしれない。【以上】

HPF コンパイラには、宣言されるプロセッサ構成の寸法の積が NUMBER_OF_PROCESSORS() の呼出しで得られる物理プロセッサ数と一致しているようなどんな PROCESSORS 宣言でも受け入れることが要求される。また、スカラのプロセッサ構成も受け入れなければならない。他の場合も同様に扱われるかどうかは、実装に依存する。

属性についての Fortran の構文との互換のため、“::” を挿入することもできる。また、その形状は次のように DIMENSION 属性として指定することもできる。

```
!HPF$ PROCESSORS :: RUBIK(3,3,3)
!HPF$ PROCESSORS, DIMENSION(3,3,3) :: RUBIK
```

Fortran と同様に、*processors-decl* の中に明示上下限並び (*explicit-shape-spec-list*) があれば、DIMENSION 属性による指定は無効になる。

```
!HPF$ PROCESSORS, DIMENSION(3,3,3) ::      &
!HPF$            RUBIK, RUBIKS_REVENGE(4,4,4), SOMA
```

ここで、RUBIK と SOMA がそれぞれ $3 \times 3 \times 3$ であるのに対し、RUBIKS_REVENGE は $4 \times 4 \times 4$ である。(しかし、前述のルールにより、このような文は完全には移植性があるとは言えない。合計の大きさが 27 である形状と 64 である形状を同時に扱うことは、HPF 処理系に必須とされていないからである。)

副プログラムから戻ると、その副プログラムに局所的に宣言されたすべてのプロセッサ構成は不定の状態になる。以下の二つのうちのどちらかの条件を満たさない限り、不定になるプロセッサ構成に対して配列やテンプレートを分散させる記述は、HPF 規格合致ではない。

- その配列またはテンプレート自身は、副プログラムから戻ることによって同時に不定となる。
- その副プログラムが呼び出されるときには、そのプロセッサ構成はいつでも同じ方法で局所的に確定し、下限値や上限値がいつも同じである。

> 【仕様の根拠】 二番目の条件は、すべての式を定数式に限るよりも少し緩やかであることに注意されたい。NUMBER_OF_PROCESSORS や PROCESSORS_SHAPE の呼出しは、この条件に違反しないため許される。【以上】

共通ブロックの変数や SAVE 属性をもつ変数は局所的に宣言されたプロセッサ構成に対してマップしてもよいが、一番目の条件を維持できない (不定にならない) ので、二番目の条件が守られなければならない。この条件は、共通ブロックの変数の慣習的な使い方、すなわち、その変数を使用したいそれぞれの有効域に同じ宣言を置く、という方法であれば共通ブロックの変数が正しく機能することを可能にしている。たとえ、その宣言が NUMBER_OF_PROCESSORS の戻り値に依存するようなプロセッサ構成に対するマッピングであってもそうである。(共通ブロックの変数のマッピングについてのより詳しい情報は 3.8 節を参照されたい。)

> 【実装者への助言】 プログラムが実行される物理プロセッサの数を翻訳時に利用者が指定する方法があるとよい。これは例えば、実装依存の指示文によるか、プログラミング環境 (例えば UNIX コマンドライン引数) を通して指定するようにできる。このような機能は HPF 仕様の範囲を越えているが、検討材料として以下に解説のための仮説的な例を示す。
>
> ```
> !ABC Corporation のマルチプロセッサのための宣言
> !ABC$ PHYSICAL PROCESSORS(8)
> !XYZ Incorporated の MPP のための宣言
> !XYZ$ PHYSICAL PROCESSORS(65536)
> !PDQ Limited のハイパーキューブマシンのための宣言
> !PDQ$ PHYSICAL PROCESSORS(2,2,2,2,2,2,2,2,2,2)
> !TLA GmbH の二次元グリッドマシンのための宣言
> !TLA$ PHYSICAL PROCESSORS(128,64)
> !上記のうちの一つがこれに影響を与えるかもしれない
> !HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
> ```
>
> さらに加えて、PROCESSORS 文で宣言されるプロセッサ構成から実行ハードウェアの物理プロセッサへの正確なマッピングを、利用者が指定する方法があるとよい。これも、実装依存の指示文によるか、プログラミング環境 (例えば UNIX コマンドライン引数) を通して指定するようにできる。このような機能は HPF 仕様の範囲を越えているが、検討材料として以下に解説のための仮説的な例を示す。

```
!PDQ$ PHYSICAL PROCESSORS(2,2,2,2,2,2,2,2,2,2,2,2,2,2)
!HPF$ PROCESSORS G(8,64,16)
!PDQ$ MACHINE LAYOUT G(:GRAY(0:2),:GRAY(6:11),:BINARY(3:5,12))
```

G の最初の次元がハイパーキューブの座標軸 0, 1, 2 を Gray-code ordering で使用し、2 番目の次元がハイパーキューブの座標軸 6 から 11 を Gray-code ordering で使用し、3 番目の次元がハイパーキューブの座標軸 3, 4, 5 と 12 を binary ordering で使用するという指定である。【以上】

3.7 TEMPLATE 指示文

TEMPLATE 指示文は、一つまたは複数のテンプレートを宣言し、それぞれの名前と次元数と各次元の寸法を指定する。この指示文は、有効域の宣言部にだけ現れることができる。

F95:14.1.2 の用語によれば、テンプレートは類 (1) の局所要素である。したがって、テンプレートは、同じ有効域の中で、変数や名前付き定数や内部手続などと同じ名前であってはならない。テンプレートの名前は、F95:12.1.2.2.1 のリストの中の他の名前と同じように親子結合と参照結合の規則に従う。

モジュール内で宣言されたテンプレートは、そのモジュールの暗黙の参照許可属性をもつ。

【仕様の根拠】 テンプレートの名前は、HPF の第一種言語要素ではなく指示文の中にだけ現れなければならないため、参照許可宣言文 (PRIVATE と PUBLIC) に現れることができない。もしも指示文が構造化された注釈ではなく通常の Fortran の文であるなら、プロセッサ構成の参照許可は参照許可宣言文に名前を指定することによって制御できるようにするのがよいであろう。【以上】

テンプレートは単純に、インデックス付けされた位置を表す抽象的な空間である。それは、(例えば「整数の配列」と同じような) 「nothing の配列」であると考えることができる。テンプレートは抽象的な *align-target* として使用することができ、そして分散することができる。

H331 *template-directive* **is** TEMPLATE *template-decl-list*

H332 *template-decl* **is** *template-name* [(*explicit-shape-spec-list*)]

例:

```
!HPF$ TEMPLATE A(N)
!HPF$ TEMPLATE B(N,N), C(N,2*N)
!HPF$ TEMPLATE DOPEY(100,100),SNEEZY(24),GRUMPY(17,3,5)
```

“::” 構文を用いると、テンプレートの宣言と同じ *combined-directive* で分散も記述することが可能になる。その場合、その指示文で宣言されるすべてのテンプレートは、DISTRIBUTE 属性が意味をもつように次元数が同じでなければならない。DIMENSION 属性も使用できる。

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK,*) ::    &
!HPF$                                WHINEY(64,64),MOPEY(128,128)
!HPF$ TEMPLATE, DIMENSION(91,91) :: BORED,WHEEZY,PERKY
```

互いに関係の深いいくつかの配列をどこかに整列したいが、それらの配列のインデックス空間全体を張る別の配列は宣言したくないとき、テンプレートが役に立つ。例えば、以下のように、4 つの $N \times N$ 配列を、大きさ $(N+1) \times (N+1)$ のテンプレートの四隅に整列したい場合である。

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) :: EARTH(N+1,N+1)
      REAL, DIMENSION(N,N) :: NW, NE, SW, SE
!HPF$ ALIGN NW(I,J) WITH EARTH( I , J )
!HPF$ ALIGN NE(I,J) WITH EARTH( I ,J+1)
!HPF$ ALIGN SW(I,J) WITH EARTH(I+1, J )
!HPF$ ALIGN SE(I,J) WITH EARTH(I+1,J+1)
```

テンプレートはまた、仮引数のマッピングについての宣言を表明するのにも役立つ (第 4 章参照)。

配列と違って、テンプレートは COMMON 文中に書けないため、異なる有効域で宣言された二つのテンプレートは、たとえ同じ名前であっても常に別個のものである。二つのプログラム単位で同じテンプレートを参照する唯一の方法は、テンプレートをモジュール内で宣言し、二つのプログラム単位で使用することである。

テンプレートは、副プログラムの引数を通して受け渡されることはない。仮引数が整列するテンプレートは、実引数からコピーされることはあっても (4.4.2 項 参照)、実引数が整列するテンプレートとは常に別のものである。HPF の実装では、実引数が整列するテンプレートは、副プログラムの出口で呼出し前と同じになるように調整される。

副プログラムから戻ると、その副プログラムに局所的に宣言されたすべてのテンプレートは不定の状態になる。以下の二つのうちのどちらかの条件を満たさない限り、不定になるテンプレートに対して変数を整列する記述は、HPF 規格合致ではない。

- その変数自身は、副プログラムから戻ったことによって同時に不定になる。
- その副プログラムが呼び出されるときには必ず、そのテンプレートはいつも同じ方法で局所的に確定する。同じ方法とは、同じ下限値、同じ上限値、および同じ定義をもつプロセッサ構成 (3.6 節 参照) に対する同じ分散の情報 (あるなら) をもつことである。

> 【仕様の根拠】 二番目の条件は、すべての式を定数式に限るよりも少し緩やかであることに注意されたい。NUMBER_OF_PROCESSORS や PROCESSORS_SHAPE の呼出しは、この条件に違反しないため許される。【以上】

共通ブロックの変数や SAVE 属性をもつ変数は局所的に宣言されたプロセッサ構成にマップしてよいが、一番目の条件を維持できない (不定にならない) ので、二番目の条件が守られなければならない。

3.8 記憶列結合と順序結合

HPF は、並列実行性能の向上のために複数のプロセッサにまたがるデータ実体のマッピングを許している。Fortran は、COMMON 文や EQUIVALENCE 文によって結合されるデータ実体の記憶場所の相互関係と、手続の境界で実引数と仮引数が結合するときの配列要素の並び順序を規定している。これら以外には、データの場所についての言語からの制約はない。

COMMON 文と EQUIVALENCE 文は、記憶単位と記憶列という潜在的なモデルに基づいて、異なるデータの間の位置合せを制約している。

> 記憶列結合は、二つ以上の記憶列が一つ以上の記憶単位を共有したりそれを介して整列されたりする場合に起こる、二つ以上の実体の結合とする。
>
> — Fortran 規格 (F95:14.6.3.1)

記憶列結合のモデルは、線形なアドレスをもつ単一のメモリであり、従来の単一アドレス空間や単一メモリ装置アーキテクチャに基づいている。このモデルは、データ実体の記憶領域がマップされるようなアーキテクチャでは深刻な効率低下をもたらす恐れがある。

順序結合は配列要素の並ぶ順序に関するもので、Fortran は配列式や配列要素が配列の仮引数と結合できることを要求している。

> 実引数の次元数および形状は、仮引数の次元数および形状と一致する必要はない。...
>
> — Fortran 規格 (F95:12.4.1.4)

記憶列結合と共に、順序結合は線形にアドレス付けされたメモリをもつシステムにおいては自然な概念である。

FORTRAN 77 コードの移植の便宜のため、HPF は順序結合と記憶列結合に依存するプログラムコードを意味あるものとして受け入れる。しかし、既存の FORTRAN 77 コードに対しては多少の修正は必要かもしれない。本章では、HPF のデータマッピングと順序結合/記憶列結合の関係を説明する。

3.8.1 記憶列結合

3.8.1.1 定義

1. 共通ブロックは**順序的**か**非順序的**のどちらかである。この区別は、明示的な指示文、または、コンパイラの暗黙値によって決定される。順序的な共通ブロックは、一つの共通ブロック記憶列 (F95:5.5.2.1) となる。
2. **結合変数グループ**は、個々の記憶列が単一の記憶列の一部となっている変数の集まりである。
   EQUIVALENCE 文、または EQUIVALENCE 文と COMMON 文の組合せによって結合づけられた変数は、結合変数グループを形成する。順序的な共通ブロックの変数は、一つの結合変数グループを形成する。
3. 結合変数グループの**大きさ**は、その記憶列 (F95:14.6.3.1) の記憶単位の数である。
4. データ実体は**順序的**か**非順序的**のどちらかである。データ実体が**順序的**である必要十分条件は、以下のいずれかが成り立つことである。

(a) 順序的な共通ブロック内に現れる。
(b) 結合変数グループに属する。
(c) 大きさ引継ぎ配列である。
(d) その型は連続型である。
(e) 順序的データ実体の部分実体である。
(f) HPF の SEQUENCE 指示文によって順序的であることを宣言されている。

順序的なデータ実体は記憶列結合と順序結合が可能であるが、非順序的なデータ実体では許されない。

5. 共通ブロックは**共通ブロック成分**の並びであり、共通ブロック成分は、一つの結合変数グループか、結合変数グループに属さない一つの変数のどちらかである。順序的な共通ブロックは単一の共通ブロック成分から成る。非順序的な共通ブロックは複数の共通ブロック成分を含むことができ、それぞれの共通ブロック成分は、順序的な変数、または、結合変数グループ、または、非順序的な変数である。

3.8.1.2 定義を説明する例

```
!例 1:
      COMMON /FOO/ A(100), B(100), C(100), D(100), E(100)
      DIMENSION X(100), Y(150), Z(200)
      EQUIVALENCE ( A(1), Z(1) )
!共通ブロック成分は (A, B), C, D, E の四つ
!大きさはそれぞれ 200, 100, 100, 100

!例 2:
      COMMON /FOO/ A(100), B(100), C(100), D(100), E(100)
      DIMENSION X(100), Y(150), Z(200)
      EQUIVALENCE ( A(51), X(1) ) ( B(100), Y(1) )
!共通ブロック成分は (A, B, C, D), E の二つ
!大きさはそれぞれ 400, 100

!例 3:
      COMMON /FOO/ A(100), B(100), C(100), D(100), E(100)
      DIMENSION X(100), Y(150), Z(200)
!HPF$ SEQUENCE /FOO/
!共通ブロック成分は (A, B, C, D, E) 一つ
!大きさは 500
```

共通ブロック/FOO/は例 1、例 2 では非順序的である。括弧で囲まれた共通ブロック成分は、結合変数グループを表している。

3.8.2 SEQUENCE 指示文

SEQUENCE 指示文は、データ実体または共通ブロックが順序的であるということを利用者がコンパイラに対して明示的に宣言できるようにする、という目的で定義されている。(共通ブロックは暗黙的には非順序的である。データ実体は 3.8 節の定義 4 が成り立たない限り非順序的である。) SEQUENCE 指示文が暗黙的に適用されるようにする翻訳環境オプションを提供している実装もある。そのような環境オプションに対応するため、HPF では NO SEQUENCE 指示文を定義している。利用者は、その有効域全体に対して、または有効域の中で選択したデータ実体と共通ブロックに対して、非順序的が暗黙に適用されるよう指定できる。

H333 *sequence-directive* **is** SEQUENCE [[::] *association-name-list*]
or NO SEQUENCE [[::] *association-name-list*]

H334 *association-name* **is** *object-name*
or / [*common-block-name*] /

制約: いかなる有効域においても、データ実体の名前と共通ブロック名は、*sequence-directive* の中に高々一度しか現れてはならない。

制約: *association-name-list* のない *sequence-directive* は、同じ有効域の中で一つしか許されない。

順序的ポインタは、順序的データ実体とだけ結合することができる。非順序的ポインタは、非順序的データ実体とだけ結合することができる。

3.8.2.1 記憶列結合の規則

1. *association-name-list* のない *sequence-directive* は、その有効域中の明示的にマップされていないデータ実体と共通ブロックのうち、他の文脈から順序的とも非順序的とも決定することができないすべてのものを指定しているかのように解釈される。
2. 順序的なデータ実体は明示的にマップできない。
3. Fortran の SEQUENCE 属性をもつ構造型の構造体の成分を明示的にマップしてはならない。構造体成分は HPF では明示的にマップできないので、この規則は公認拡張仕様の下でだけ適用されることに注意されたい。 ⇓
4. 共通ブロックが非順序的のとき、以下のすべてを満たさなければならない。
   (a) 同じ共通ブロックが複数出現するとき、すべての出現で共通ブロック成分の数は同じでなければならず、対応する成分はそれぞれ同じ大きさの記憶列でなければならない。
   (b) 共通ブロックのある出現で共通ブロック成分が非順序的変数であるとき、すべての出現でその共通ブロック成分は非順序的でなければならず、型、形状、およびマッピング属性が同一でなければならない。
   (c) 共通ブロックはすべての出現で非順序的でなければならない。

3.8.2.2 記憶列結合についての議論

【利用者への助言】 ここで述べた規則によって、共通ブロックが宣言されているすべての有効域でその共通ブロック成分が等価であるなら、共通ブロック中の変数をマップ

することができる。

正しいFortranプログラムであっても、HPFでは修正なしでは必ずしも正しいプログラムにはならない。共通ブロックと同時にEQUIVALENCEを使用すると、データのマッピング可能性に複雑な影響を与える。性能的な最適化を促進するため、データ実体についてマップ可能と考るのがHPFの暗黙の規定である。異なる有効域では結合変数グループが一致しない共通ブロックを使用する副プログラムについて、正しく分割翻訳ができるようにするためには、HPFのSEQUENCE指示文を挿入する必要がある。

データ実体と共通ブロックの状態を知るためのチェックリストを利用者のために用意した。以下の質問に順に当てはめてみてほしい。

- そのデータ実体は、順序的か非順序的かを指示する何らかの明示的な言語の文脈の中に現れるか？(例えば、EQUIVALENCE文に現れれば順序的。)
- そうでなければ、そのデータ実体は明示的なマッピングの指示文に現れるか？
- そうでなければ、そのデータ実体または共通ブロック名はSEQUENCE指示文かNO SEQUENCE指示文の名前のリスト中に現れるか？
- そうでなければ、その有効域は名前なしのSEQUENCEかNO SEQUENCEを含んでいるか？
- そうでなければ、その名前を暗黙にSEQUENCEにするような何らかの実装依存の環境下で翻訳しているか？
- そうでなければ、コンパイラはそのデータ実体または共通ブロック名を非順序的と考え、Fortranの順序結合と記憶列結合を無視したデータマッピングの最適化を行ってよい。

【以上】

【実装者への助言】利用者を保護し古いコードの可搬性を促進するために、二つの付加機能を強く推奨する。第一に、処理系が提供するべきものは、順序的と宣言されていない共通ブロックについて、すべてのマップ可能な配列の型と形状、そして共通ブロック内の結合変数グループの大きさが、すべての有効域で一致しているかどうかを検証する機能である。この一致性の検証では、共通ブロック中の変数に同一のマッピングが指定されているかどうかも検出してほしい。手続間の情報を参照しない実装の場合には、リンク時のチェックでも可能である。二番目の付加機能として推奨するのは、データ実体と共通ブロックについて、順序的を暗黙と考えて翻訳することを宣言する機能である。この機能の目的は、記憶列結合があることが分かっている巨大な古いライブラリや副プログラムについて、HPFのSEQUENCE指示文を全部の共通ブロックに記述する修正を必要としないで翻訳ができるようにすることである。【以上】