第5章　INDEPENDENT指示文及び関連の指示文

HPF の INDEPENDENT 指示文の目的は、プログラマがコンパイラに並列化のための追加情報を与えられるようにすることである。ユーザは、DO ループの中に、ループのある繰返しにおいて定義され、別の繰返しにおいて使用 (読み出しまたは書き込み) されるようなデータが存在しないことを表明できる。同様の情報を、FORALL 文のインデックス値の組み合わせに対しても与えることができる。このような情報は、コンパイラの最適化に役立つことがある。しかし、このような情報を知るには、プログラマにしか解らないアプリケーションに対する知識を必要とする場合もあるだろう。そこで、HPF では、ユーザがこれらの表明を行なうことを許し、コンパイラは、それらの表明に頼ってコンパイルを行なうことができる。表明が正しければ、プログラムは意味的に変更されない。もし表明が間違っている場合、そのプログラムは HPF 規格合致でなく、その意味は不定となる。

HPF1.0 と比較すると、HPF2.0 の INDEPENDENT 指示文では、INDEPENDENT ループの中で集計演算を行なうことが許されている。ただし、その集計演算子は、組込みであり、かつ結合則、交換則を満たす Fortran の演算子 (例 .AND.) または関数 (例 MAX) でなければならない。このような演算を含むデータ並列計算では、1 つまたはそれ以上の変数が、複数のループ繰返しによって更新されることが多いために、HPF1.0 では INDEPENDENT ループとして表現できない場合がしばしばあった。しかし、そのような場合でも、変数の更新の順序が最終的な結果に影響しない場合には、並列性が引き出せる可能性があり、また、そうしたいと望まれる場合もあるだろう。次のようなループで表される総和演算はこの典型的な例である。

```
DO I = 1, 1000000000
    X = X + COMPLICATED_FUNCTION(I)
END DO
```

このループは、共有変数 X が、ループのそれぞれの繰返しによって、不可分な方法で (in an atomic manner) 更新される限り、並列に実行可能である。あるいは、更新を一時的なローカルアキュムレータ変数に対して行ない、(短い) 最終フェーズでそれら変数の値を X の初期値とマージすることで、ループを並列に実行することができる。どちらのケースにおいても、計算は本質的に並列であるが、HPF1.0 での INDEPENDENT の厳密な定義からすると許されない。

ここで、現在の Fortran がデータ並列計算を表現するいくつかの手段を含んでいることについて触れるのは価値があるだろう:

- 配列代入文 (WHERE 文を含む)
- 組込み手続や利用者定義手続の要素別処理

- FORALL 文及び FORALL 構文 (PURE 関数の要素別処理を含む)
- SUM や TRANSPOSE などの変形組込み関数

FORALL と PURE は、HPF1.0 から Fortran に採用された。これらは全て現在 Fortran の一部となっているので、このドキュメントであらためてとりあげることはない。

5.1 INDEPENDENT 指示文

INDEPENDENT 指示文は、インデックス付き DO ループまたは、FORALL 文の前に置くことができる。これは、コンパイラに対し、後続の DO ループの繰返しや、後続の FORALL 文の中の演算を独立に (つまり、任意の順序で、またはインターリーブに、または並列に) 実行でき、そうしてもプログラムの意味は変らないことを表明する。

INDEPENDENT 指示文は、DO ループまたは FORALL の前に置かれ、それらの動作を表明する。このとき、INDEPENDENT 指示文は、そのループまたは FORALL へ**適用される**と言う。INDEPENDENT 指示文の構文は次のとおりである。

H501 *independent-directive* **is** INDEPENDENT [, *new-clause*]
[, *reduction-clause*]

H502 *new-clause* **is** NEW (*variable-name-list*)

H503 *reduction-clause* **is** REDUCTION (*reduction-variable-list*)

H504 *reduction-variable* **is** *array-variable-name*
or *scalar-variable-name*
or *structure-component*

制約: *independent-directive* の後の最初の非注釈行は、*do-stmt*、*forall-stmt*、または *forall-construct* でなければならない。

制約: *independent-directive* の後の最初の非注釈行が *do-stmt* である場合、その文は *do-variable* を含む *loop-control* オプションを持たなければならない。

制約: NEW 節または REDUCTION 節がある場合、指示文の後の最初の非注釈行は *do-stmt* でなければならない。

制約: NEW 節または REDUCTION 節内に指定する *variable* 及びそれらの成分及び要素は、次のものであってはならない。

- 仮引数
- SAVE 属性または TARGET 属性を持つもの
- COMMON ブロックに現われるもの
- EQUIVALENCE 文により他の実体と記憶列結合するもの
- 参照結合したもの
- 親子結合したもの

- 親子結合により他の有効域でアクセスされるもの

制約: *reduction-variable* として現われる変数は、同じ *independent-directive* の *new-clause* 中に現われてはならず、*independent-directive* が適用される後続の *do-stmt*、*forall-stmt* 及び *forall-construct* の範囲内 (すなわち、ソース上でのループ本体部) の *new-clause* 及び reduction-clause に現われてはならない。

制約: *reduction-variable* 中の *structure-component* は、*subscript-section-list* を含んではならない。

制約: *reduction-variable*[1] として現われる変数は、組込み型でなければならない。また、CHARACTER 型であってはならない。

【仕様の根拠】 2 番目の制約は、INDEPENDENT 指示文が、WHILE ループや、単純な DO ループ (すなわち、「無限ループ」) には適用できないことを意味している。そのようなケースにおける INDEPENDENT は、0 または 1 回の繰返しからなるループにのみ適用できるという案もあった。しかし、それにより生じうる混乱のほうが、得られるであろう利益よりも重大だと思われた。【以上】

INDEPENDENT 指示文が DO ループに適用された場合、ループの繰返しが、直接的にも間接的にも、他の繰返しに干渉する可能性がないことの、プログラマによる表明となる。干渉するとは、次のような操作として定義される。

- 同じ不可分な実体へ代入を行なう二つの演算は、互いに干渉する。(部分実体を含まないデータ実体は、不可分と呼ばれる。)
  - 例外: 変数が NEW 節に現われる場合、DO ループの異なる繰返しでのその変数に対する代入は互いに干渉**しない**。この理由は、第 5.1.2 項で説明されている。
  - 例外: 変数が REDUCTION 節に現われる場合、DO ループの範囲内の集計文による代入と、同じループ内の他の集計文による代入とは互いに干渉しない。この理由は、第 5.1.3 項で説明されている。

  実体への代入という操作は以下のものを含む。

  - 代入文は、左辺と、その全ての部分実体に対して代入を行なう
  - ASSIGN 文は、その整数変数に対して代入を行なう
  - STAT=指定子を伴う ALLOCATE と DEALLOCATE 文は、STAT 変数への代入を行なう
  - DO 文は、そのインデックスへの代入を行なう
  - IOSTAT=指定子を伴う入出力文は、IOSTAT 変数への代入を行なう。また、以下で述べるようにその他の実体への代入を行なう場合もある。
  - 非同期の READ 及び WRITE 文は、(第 10 章で述べられているように) その ID=変数に代入を行なう。

[1] 原文は *reduction-var*

- READ 文は、その入力変数並びの全ての変数と、NAMELIST を通して実行時に参照される全ての変数に対する代入を行なう。SIZE=指定子を伴う READ 文は、その SIZE 変数に代入を行なう。
- INQUIRE 文は、その指定子並びの全ての変数に対する代入を行なう。ただし、UNIT 指定子及び FILE 指定子は除く。
- 複合文 (例えば IF 文) は、それを構成する文が実体に対する代入を行なう場合に、代入を引き起こす。
- 副プログラムの実行は、その副プログラムで実体に対する代入が行なわれる場合に、代入を引き起こす。

- 不可分な実体に代入を行なう演算は、その実体の値を使用する演算と互いに干渉する。
  - 例外: 変数が NEW 節に現われる場合、DO ループのある繰返しでのその変数に対する値の代入は、他の繰返しにおけるその変数の使用とは干渉しない。この理由は、第 5.1.2 項で説明されている。
  - 例外: 変数が REDUCTION 節に現われる場合、DO ループの範囲内の集計文によるその変数への代入は、同じループ内の集計文で許可されるその変数の使用とは干渉しない。この理由は第 5.1.3 項で説明されている。

  変数の値を計算する式は、その実体を使用する。これには、代入文の右辺での使用、代入文の左辺の添字での使用、条件式、入出力文の指定子並び、WRITE 文の出力並び、ALLOCATE 文の割付け形状指定子、及びこれに類する状況での使用を含む。

  【仕様の根拠】 これらは、並列実行を可能にするための古典的な Bernstein の条件である。一つの変数への「同じ値」の 2 回の代入は互いに干渉し合い、したがってそのような代入文を含む INDEPENDENT ループは HPF 規格合致でないことに注意されたい。これが許されない理由は、一部のハードウェアではそのような重複した代入をサポートするのが困難であることと、上記の定義のほうが、概念上、より明確であると感じられたことである。同様に、同じ位置への複数の値の代入を INDEPENDENT として表明することは、たとえ論理的にそのプログラムが任意の可能な値を受け入れる場合でも、HPF 規格合致ではない。この場合、論拠が「概念上、より明確である」という点でも、非決定論的な動作を避けたいという点でも、上記の解決策が選択された。【以上】

- ALLOCATE 文、DEALLOCATE 文、NULLIFY 文、及びポインタ代入文は、同じポインタに対する他の参照、ポインタ代入、ALLOCATE、DEALLOCATE 及び NULLIFY と干渉しあう。加えて、ALLOCATE 及び DEALLOCATE 文は、ALLOCATE 及び DEALLOCATE された実体に対する使用または代入と干渉し合う。

  【仕様の根拠】 これらの条件は、Bernstein の条件をポインタに対して拡張したものである。Fortran のポインタは、第一種のデータ型ではなく、実体または部分実体に対する別名なので、他の変数よりも少し厳密な定義が必要となる。【以上】

- ループ本体の外にある分岐先の文への制御の移動は、ループ内にあるその他の全ての演算との間に干渉を発生させる。

- EXIT、STOP、PAUSE のいずれかの文の実行は、ループ内にあるその他の全ての演算との間に干渉を発生させる。

  【仕様の根拠】 分岐 (GOTO によるものか、入出力文での ERR=による分岐) は、ループの繰返しが一部実行されないことを意味し、それらの計算との間に重大な干渉を発生させる。同じことが EXIT とその他の文にも言える。これらの条件は、INDEPENDENT ループ内の手続呼出しを制限していないことに注意されたい。ただし、そのような呼出しが、STOP 文や PAUSE 文によってループの外にある文に選択戻りを行うことは禁止されている。【以上】

- 同じファイルまたは装置に接続した、INQUIRE を除く二つのファイル入出力演算は、互いに干渉する。二つの INQUIRE 演算は、互いに干渉しない。ただし、INQUIRE 演算は、同じファイルに接続したその他の入出力演算との間に干渉を発生する。

  【仕様の根拠】 Fortran では、データ転送文またはファイル位置付け文の後のファイル位置が入念に定義されているので、これらの演算はプログラムの大域的な状態に影響を及ぼす。(直接探査ファイルについてさえ、ファイル位置が定義されていることに注意されたい。) 複数の停留データ転送文は、一つの変数へ同じ値を複数回代入するのと同様に、ファイル位置に影響を及ぼし、同じ理由から禁止されている。複数の OPEN 演算と CLOSE 演算は、ファイルと装置の状態に影響を及ぼし、これもまた大域的な副作用を持つ。INQUIRE はファイルの状態に影響を及ぼさず、したがって他の問合せに影響を及ぼさない。しかし、その他のファイル演算は、INQUIRE によって報告された性質に影響を及ぼす場合がある。【以上】

- 副プログラムの呼出しで生じるデータの再整列または再分散 (第 4 章を参照) は、同じデータへの参照や、同じデータの別の再マッピングとの間に干渉を発生する。

  【仕様の根拠】 再マッピングは、特定の配列要素を格納しているプロセッサを変更する場合があり、それによって、その要素の代入または使用に干渉が発生する。これは、手続の呼出しから戻った時点でその再マッピングが「元に戻される」場合でも当てはまる。手続を呼び出して実行している間、配列要素の格納される位置は変更される。よって、呼出し側での参照、同じ手続の別の実行における参照、他の手続呼出しによる配列の再マッピングと干渉し合う。【以上】

  【利用者への助言】 この規則により、公認拡張機能の REALIGN 及び REDISTRIBUTE によるデータの再マッピングも、干渉を引き起こす。詳細は第 8.5 節を参照されたい。【以上】

FORALL の場合の INDEPENDENT の解釈は、DO の場合と同様である。つまりそれは、INDEPENDENT が適用される FORALL のインデックスのどのような組合せも、別の組合せによって読み取られる不可分な記憶単位への代入を行わないことを表明する。同じ本体を持つ

DO と FORALL は、その両方が INDEPENDENT 指示文を持つ場合には等価である。これは第 5.1.1 項に例示されている。

INDEPENDENT ループまたは FORALL の中から手続が呼び出される場合、その手続中の全ての局所変数は、SAVE 属性を持たない限り、それぞれの呼出しに対して独立であるとみなされる。これは、Fortran の規格とも合致する。それゆえ、異なる繰返しでの呼出しにおける局所変数の使用は、上記で定義したような干渉を引き起こさない。

【実装者への助言】 適切な Fortran の実装では、手続の呼出しごとに局所変数に異なる記憶域を与えることを避けることがよくある。同じことが、HPF の実装においてもあてはまる。しかしながら、そのような実装においても、INDEPENDENT 指示文は、同じように解釈される必要がある。もし、局所変数が呼出し毎に固有の記憶域を割り当てられないのであれば、これらの意味論を尊重してその INDEPENDENT ループは逐次化される必要がある (あるいは参照の衝突を避けるために別のテクニックを用いる必要がある)。【以上】

これらはすべて、干渉についての説明であり、特定の構文を禁止したものでないことに注意されたい。これらの制限事項に一つ以上違反すると思われる文でも、制御フローの上で実行されないなら、INDEPENDENT ループの中に入れられる。これらの制限事項は、計算資源が使用可能なら、INDEPENDENT ループが安全に並列実行されることを許す。この指示文は純粋に助言的なものであり、コンパイラは、その情報を有効利用できない場合には、自由にそれを無視できる。

【実装者への助言】 これらの制限により、INDEPENDENT ループを安全に並列化して実装できるが、このことは全てのアーキテクチャあるいは全てのプログラムに対して有益であるとは (あるいは可能であるとは) 限らない。例えば、

- INDEPENDENT ループから、明示的にマップされた局所変数を持つルーチンが呼び出されるかもしれない。これに対する実装は、マッピングを実装するか (実装方法によっては呼出しを逐次化することが必要となるだろう)、明示された指示文を無視する (これはユーザを驚かせることになるかしれない) 必要がある。
- INDEPENDENT ループは、その繰返し毎に非常に異ったふるまいを示すかもしれない。例えば、

```
!HPF$ INDEPENDENT
      DO i = 1, 3
        IF (i.EQ.1) CALL F(A)
        IF (i.EQ.2) CALL G(B)
        IF (i.EQ.3) CALL H(C)
      END DO
```

  この例は、SIMD 型の計算機における実装で明らかに問題となる。

- INDEPENDENT ループ中から呼ばれたサブルーチンで、グローバルにマップされたデータに対する参照があるかもしれない。分散記憶型の計算機では、このデータを参照するために通信を生成することは非常に挑戦的なことであるかもしれない。

なぜなら、一般的には、データの実際の持ち主も同様にそのサブルーチンを呼ぶという保証はないからである。

いずれにせよ、正しい振舞いをさせるのは、実装側の責任であり、それは最適化を制限することになるかもしれない。実装は、INDEPENDENT による表明が無視される可能性がある場合は、なんらかのフィードバックを返すことが推奨される。【以上】

5.1.1 INDEPENDENT 指示文の視覚化

```
DO i = 1, 3
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```

```
FORALL ( i = 1:3 )
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END FORALL
```

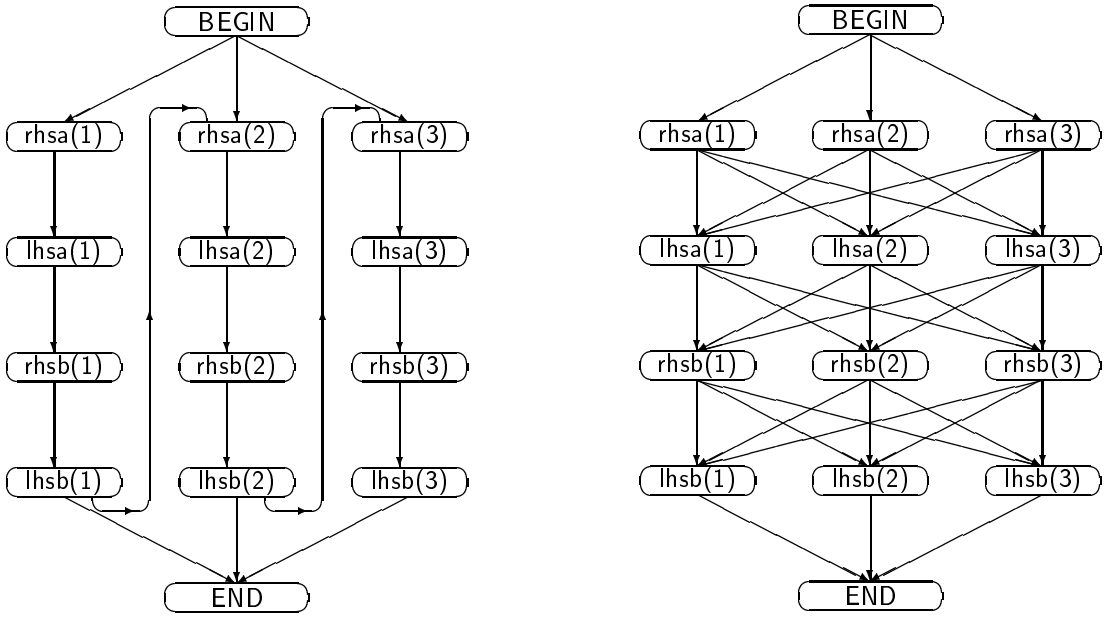
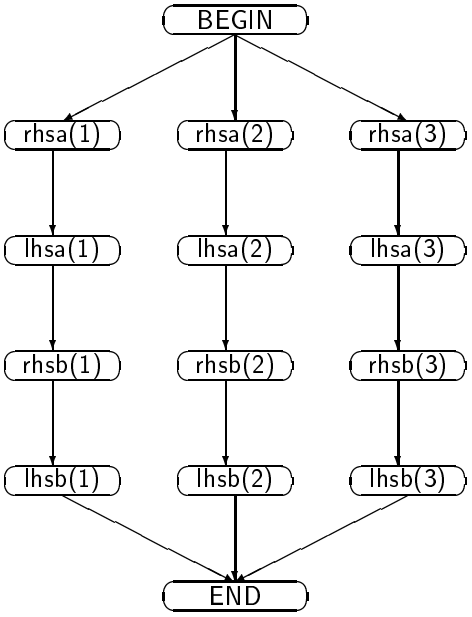


図 5.1: INDEPENDENT 表明を伴わない DO と FORALL の依存関係

INDEPENDENT 指示文は、図形的にはプログラムを表す先行制約グラフからエッジを除去したものとして視覚化できる。図 5.1 は、DO と FORALL の中に通常存在する可能性がある依存関係の一部を示している。(中間的な依存関係のほとんどは示されていない。) 左辺のノード (例えば、“lhsa(1)”) から右辺のノード (例えば、“rhsb(1)”) への矢印は、右辺の計算が、左辺のノードで代入された値を使用する可能性があることを意味する。したがって、右辺は左辺が格納を完了した後に計算されなければならない。同様に、右辺のノードから左辺のノードへの矢印は、左辺が右辺の計算に必要な値を上書きする可能性があることを意味し、この場合も順序付けが強制される。BEGIN ノードからのエッジと END ノードへのエッジは、制御の依存関係を表している。INDEPENDENT 指示文は、コンパイラが強制する必要がある唯一の依存関係は図 5.2 に示す依存関係であることを表明する。つまり、INDEPENDENT を使用するプログラマは、コンパイラがこれらのエッジだけを強制するならば、結果としてのプログラムはすべてのエッジが存在するプログラムと等価になることを保証していることになる。表

```
!HPF$ INDEPENDENT
DO i = 1, 3
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```




```
!HPF$ INDEPENDENT
FORALL ( i = 1:3 )
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END FORALL
```

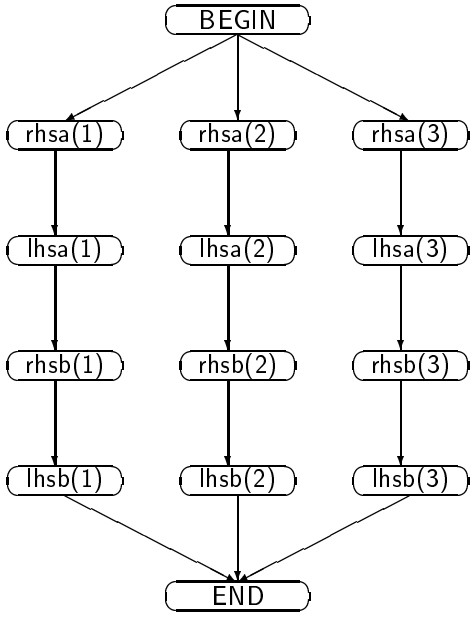


図 5.2: INDEPENDENT 表明を伴う DO と FORALL の依存関係

明された依存関係の集合は、INDEPENDENT な DO 構文と FORALL 構文とで同一であることに注意されたい。

コンパイラは、これらの表明の一つが正しくないことを証明できる場合には、警告を生成することが許される。ただし、生成しなくてもかまわない。そのような偽の表明が入っているプログラムは HPF 規格合致でなく、したがって HPF では定義されておらず、コンパイラは、適当と考えられる任意の処置が取れる。

5.1.1.1 INDEPENDENT の例

```
!HPF$ INDEPENDENT
DO i = 2, 99
  A(I) = B(I-1) + B(I) + B(I+1)
END DO
```

これは、INDEPENDENT ループの最も単純な例の一つである。(説明を簡単にするため、この項のすべての例は、コード内に使用されているどの変数の間にも記憶列結合や順序結合がないことを仮定している。) 各繰返しにおける代入は、配列 A 内の異なる位置に対して行われ、それによって前述の最初の条件が満たされる。右辺には A の要素が使用されていないので、ループ内で代入されるどの位置も読み取られず、それによって 2 番目の条件が満たされる。しかし、B の多数の要素が繰り返し使用されることに注意されたい。これは、INDEPENDENT の定義によって許される。この例では、この表明は関与する変数の値に関係なく真である。

```
!HPF$ INDEPENDENT
FORALL ( I=2:N ) A(I) = B(I-1) + B(I) + B(I+1)
```

この例は、すべての点で最初の例と等価である。

```
!HPF$ INDEPENDENT
DO I=1, 100
  A(P(I)) = B(I)
END DO
```

この INDEPENDENT 指示文は、配列 P の要素が重複する値を全く持たないことを表明する (さもないと、A が代入されたときに干渉が起きる)。したがって、DO ループは次の Fortran の文と等価である。

```
A(P(1:100)) = B(1:100)
```

5.1.2 NEW 変数

NEW 節は、指定された変数が、INDEPENDENT ループの各繰り返しに対してプライベートな変数であることを表明する。すなわち、DO ループの各繰返しで、NEW 変数に対して新しい実体が作られ、各繰返しの最後にそれらの実体が破壊されるならば、ループ中でのそれらに対する代入および使用には干渉関係がなく、よってプログラムの振舞いには変化がないということを意味する。それゆえ、いかなる値もループの前の文から NEW 変数に入ってくることはな

く、いかなる値も NEW 変数からループの後ろの文に出ていくことはなく、そして、(これが最も重要なのであるが) いかなる値も、ループのある繰返しから他の繰返しへ NEW 変数を通して渡っていくことはない。

【利用者への助言】 ポインタまたは割付け配列を、NEW 節中に記述できる。この場合の前段落の解釈は、ループの入口でのそのような変数の値、結合状態、割付け状態を信頼してはならず、そのような変数は、使用される前にループ本体中で割り付けられるか、ポインタ代入されなければならないということである。さらに、そのような変数は、その使用が終わった際にループ本体中で DEALLOCATE または NULLIFY することが望ましい。【以上】

【仕様の根拠】 NEW 変数は、INDEPENDENT ループの中でテンポラリを宣言する手段となる。この機能がなかったとすると、多数の概念的に独立したループは、上記のかなり厳しい要件を満たすために実質的な書き換え (スカラから配列への拡張も含む) を必要とする。テンポラリは、それが割り付けられる構文上最も内側のレベルでだけ NEW として宣言すればよいことに注意されたい。なぜなら、囲んでいるすべての INDEPENDENT 表明は、その NEW を考慮に入れなければならないからである。また、入れ子になった DO ループ用のインデックス変数を NEW として宣言しなければならないことにも注意されたい。別な案として、インデックス変数の有効範囲をそのループ自体に限定する方法もあったが、それでは Fortran の意味論が変更される。ただし、FORALL の意味論によって制限されている FORALL インデックスは、NEW 宣言を必要としない。【以上】

5.1.2.1 NEW の例

```
!HPF$ INDEPENDENT, NEW(I)
DO I = 1, 10
   A(I) = B(I-1)
END DO
```

この例は、NEW 節があってもなくても正しいものとなる。どちらの場合にも、コンパイラは確信を持って配列 A に対する代入文を並列化できる。しかし、NEW 節は、それに加えて、ループインデックス I がループの実行後にも使用されないということを表明する。コンパイラはこの情報により、他のプロセッサ上にある I の複製を更新する手間を省けるし、他の最適化が可能になることもある。

```
!HPF$ INDEPENDENT, NEW (I2)
DO I1 = 1,N1
   !HPF$ INDEPENDENT, NEW (I3)
   DO I2 = 1,N2
      DO I3 = 2,N3   ! 内側ループは独立ではない!
          A(I1,I2,I3) = A(I1,I2,I3) - A(I1,I2,I3-1)*B(I1,I2,I3)
      END DO
   END DO
END DO
```

最内側のループは、A の各要素が、ループの前回の繰返しで定義された値を使用して計算されているので独立ではない。しかし、外側の二つのループは、A の異なる要素を参照しているので、独立している。内側のループのインデックスは、最も外側のループの異なる繰返しの中で代入及び使用されるので、NEW 節が必要になる。

5.1.3 REDUCTION 変数と REDUCTION 文

REDUCTION 節は、指定された変数が、INDEPENDENT ループ内で、交換則、結合則を満たす一連の演算によって更新されることを表明する。更に、REDUCTION 変数の中間的な値がループ中で、(もちろんその変数自体の更新を除いて) 使用されないということも表明する。それゆえ、ループ後の REDUCTION 変数の値は、集計ツリーの結果値となる。

> 【仕様の根拠】REDUCTION 変数は、INDEPENDENT ループ内で計算される値の累積値を求める手段となる。もしこの機能が無ければ、プログラマは、更新された情報を、ループの繰返し回数と同じだけの大きさのテンポラリ配列にためておき、ループの後で、組込みの集計関数や XXX_SCATTER ライブラリ関数を使用しなければならない。この方法の問題点は、テンポラリ配列の大きさが極端に大きくなる可能性があることである。【以上】

集計の意味論については、第 5.1.4 項で詳しく論じられている。この節では、正確な構文を定義する。

reduction-variable として指定された変数は、その直後の DO ループが活動状態 (実行されている) である間、**保護されている**と言う。その変数は、それが保護されているループが活動状態である間は、次のただひとつの例外を除いて、参照されてはならない。その変数は、ある特別な形の代入文中の特別な位置にのみ現われることができる。この代入文は、ループの範囲内に (ソース上の本体部に) なければならない。特に、その変数は、いかなる HPF 指示文 (NEW 節の変数リストを含む) の中にも現われてはいけない。これには、同じ INDEPENDENT 指示文中の全ての NEW 節が含まれる。

集計文 (reduction statement) とは、集計変数 (reduction variable) が指定された REDUCTION 節が対象とする INDEPENDENT DO ループの範囲内にある、次のような特別な形をした代入文である。この記述は、HPF の文法の一部ではないが、集計変数が許される代入文の厳密な定義をするために用意されたものである。

H505 *reduction-stmt*

is *variable* = *variable mult-op mult-operand*
or *variable* = *add-operand* * *variable*
or *variable* = *variable add-op add-operand*
or *variable* = *level*-2-*expr* + *variable*
or *variable* = *variable and-op and-operand*
or *variable* = *and-operand and-op variable*
or *variable* = *variable or-op or-operand*
or *variable* = *or-operand or-op variable*
or *variable* = *variable equiv-op equiv-operand*
or *variable* = *equiv-operand equiv-op variable*
or *variable* = *reduction-function* (*variable* , *expr*)
or *variable* = *reduction-function* (*expr* , *variable*)

H506 *reduction-function* **is** MAX
or MIN
or IAND
or IOR
or IEOR

制約: *reduction-stmt* 中に *variable* が 2 つ現われる場合、その 2 つは字面的に等しくなければならない。

第 5.1 節の最初の二つの表明は、集計文の許された位置に現われた集計変数が、INDEPENDENT ループの繰返し間に干渉を発生させないということの説明になっている。集計変数に対する別の代入もしくは参照はすべて、集計文との間に干渉を発生**する**。これには、副プログラム中や集計文の *expr* 中での参照も含まれる。

INDEPENDENT ループ中の集計文で更新される変数は、REDUCTION 節で明示的に指定されることで、保護されなければならない。この REDUCTION 節は、次のような条件を満たす独立ループのうち最も外側の独立ループに対する、INDEPENDENT 指示文になければならない。

- 集計文を含む。
- 集計変数を指定する NEW 節を持たない。かつ、
- 集計文を含み、集計変数に対する NEW 節を持つループがもしあるなら、そのようなループの最内のものの内側にある。

もし同じ変数が 2 つかそれ以上の集計文において更新される場合、それらの文の演算子は、同じクラスでなければならない (つまり、もし一方が *add-op* である場合、両方が *add-op* でなければならない)。

【利用者への助言】 集計文が実行される時には、入れ子になったいくつかの DO ループが活動状態になる。変数が更新される集計文を囲む INDEPENDENT DO ループがいくつか入れ子になっている場合、どのループに REDUCTION 節をつけるべきか。その答は、集計変数はそのループまたは内側ループの NEW 節に現れてはならない、という制約下での最外側ループである。次の例を考えてみる。

```
!HPF$ INDEPENDENT, NEW(J), REDUCTION(X)
DO I = 1, 10
    !HPF$ INDEPENDENT
    DO J = 1, 20
        X = X + J
    END DO
END DO
```

REDUCTION 節を、内側の INDEPENDENT 指示文に移動することは間違いである。なぜなら、X は、外側ループの各繰返しにおいて、集計演算により (20 回) 更新されるので、その値は、外側ループの実行が完了されるまで、最終的に定義されないからである。【以上】

reduction-stmt に現れる *variable* は、配列要素または部分配列であってよい。*reduction-stmt* に現れる二つの *variable* は、字面的に等しくなければならない。演算子の優先順位と式中での括弧の用法に関する Fortran の規則は、集計演算子が右辺での最上位の (すなわちいちばん最後に評価される) 演算子であることを保証する。それゆえ次の形の集計文は許されない。

```
X = X * A + 1
```

INDEPENDENT 指示文の文法では、REDUCTION 節の集計変数として配列要素や部分配列を指定することは許されていないことに注意されたい。そのような部分実体を集計文で使用することはできるが、集計変数として扱われるのは、配列または文字型変数の全体である。

集計演算子または関数として使用可能なものは、すべて結合則を満たす必要がある。(これは、数学的な意味においてである。一般に Fortran 言語処理系やハードウェアによる数学演算子の実装では結合則は満たされない。)

一つの集計変数に対する集計文が複数ある時、ほとんどの場合、集計変数の更新に使われる演算子はただ一つである。しかし、+と-を同時に同じ集計変数に対して使用することは可能である。数学的には、減算は、正負を逆にした値の加算と等価だからである。次の例を考える。

```
!HPF$ INDEPENDENT, REDUCTION(X)
      DO I = 1, 100
        X(IDX(I,1)) = X(IDX(I,1)) + Y(I)
        X(IDX(I,2)) = X(IDX(I,2)) - Y(I)
      END DO
```

おなじことが、乗算 (*) と除算 (/) についてもあてはまる。それ以外の演算子の混合は許されない。

【利用者への助言】 次の文、

```
X = I + X
```

は、集計文として許されるが、ほとんどの場合、

```
X = X + I
```

の方が見た目がすっきりするだろう。【以上】

5.1.4 集計の実装と意味論

HPF では、集計文を伴う INDEPENDENT DO ループの、可能な並列化法を指定しており、そのために、そのようなループの意味論を規定する。

Fortran の組込み関数 SUM の結果が、その引数配列要素の和の実装に依存した近似として定義されているように、HPF での INDEPENDENT DO ループの出口での集計変数の値も、完全には定義されない。とり得る値の一つは、そのループを逐次実行した時の結果と等しいというものだが、それ以外の、実装依存の近似値が得られる場合もあり得る。しかし、そういっ

た実装依存の値でも、必ずそのループを逐次実行した時に得られる結果の近似値になっている。もし丸め誤差や、アンダーフロー、オーバーフローといったものがなければ、逐次実行による値と等しい値が得られるだろう。

【利用者への助言】 もしオーバーフロー、アンダーフローや丸め誤差が起こる場合、それは正しいプログラムにおける HPF 指示文が、異なる結果を生じさせる数少ない場合の一つである。しかし、他のシステムでこれらの演算を並列化する場合でも、同様の問題が、同様の理由により起きるのである。【以上】

保護された集計変数に対する参照は、集計文によるものを除いては存在しないので、これらの変数が保護されている間に持つであろう値を定義しておくことは必ずしも必要ではない。

【利用者への助言】 次の「実装者への助言」は、集計文を伴う INDEPENDENT ループのふるまいを解釈するのに役立つ。【以上】

【実装者への助言】 この項での議論では、「プロセッサ」という用語は、1 台の物理的なプロセッサか、INDEPENDENT ループの一部または全部の繰返しを全員が逐次に実行している物理プロセッサの 1 グループを表すものとする。

ここで、可換な集計演算に対する単純な実装法を示す。INDEPENDENT ループの入口で、実行中の全てのプロセッサが、INDEPENDENT 指示文の REDUCTION 節で指定された変数に対応するプライベートなアキュムレータ変数を割り当て、それを、対応する組込み集計演算子の単位元と同じ値に初期化する。プライベートアキュムレータ変数は、集計変数と同じ形状、型、種別型パラメタを持つ。

組込み演算子に対する単位元を、表 5.1 に示す。

演算子	単位元
+	0
-	0
*	1
/	1
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.

表 5.1: 組込み集計演算子の単位元

集計関数として使用できる組込み関数と、その単位元を表 5.2 に示す。

各々のプロセッサは、ループの繰返しの一部分を担当し、集計文に出会うと、自分が持つアキュムレータ変数を更新する。プロセッサは、ループの繰返しを任意の順番で実行することができる。さらにいえば、ある繰返しを開始し、それを途中で中止して、別の繰返しの一部または全部を実行した後に、途中で中止した繰返しを再開することも可能

関数	単位元
IAND(I,J)	NOT(0) (全ビットが 1)
IOR(I,J)	0
IEOR(I,J)	0
MIN(X,Y)	集計変数と同じ型、種別型パラメタを持ち、絶対値が最大の正数
MAX(X,Y)	集計変数と同じ型、種別型パラメタを持ち、絶対値が最大の負数

表 5.2: 組込み集計関数の単位元

である。しかし、プライベートなアキュムレータ変数の更新は、すべて集計文の実行を通して行なわれ、集計文はそれぞれが不可分 (atomically) に実行される。

集計変数の最終的な値は、プライベートアキュムレータ変数を、ループ入口での集計変数の値と、集計演算子を用いて結合することにより得られる。この集計の順序は、組込みの集計関数 (SUM など) がそうであるのと同様に、言語処理系依存である。

例として、次を考える。

```
      REAL Z

      Z = 5.
!HPF$ INDEPENDENT, REDUCTION(Z)
      DO I = 1, 10
        Z = Z + I
      END DO
```

Z の最終的な値は、$5 + (1+2+3+4+5+6+7+8+9+10) = 60$ である。足し算が行なわれる順序は、HPF では規定されない。

二つめの例として、ADD_SCATTER を INDEPENDENT ループとして実現する例をあげる。

```
!HPF$ INDEPENDENT, REDUCTION(X)
DO I = 1, N
    X(INDEX(I)) = X(INDEX(I)) - F(I)
END DO
```

これに対する実装としてはおそらく、各プロセッサが X と同じ型で同じ形状を持つアキュムレータ配列 XLOCAL のプライベートな複製を持ち、それをゼロに初期化する。各繰返しでは、XLOCAL(INDEX(I)) から F(I) の値を引き算する。最終的な結果を得るために、この実装では、全てのプライベートアキュムレータ配列と、X の初期値を結合する必要がある。結合に使用する演算子は、集計の演算子と同じもの、すなわち加算の演

算子であり、結果は、X の初期値とアキュムレータ配列の和になる。この実装には、ローカルなアキュムレータの内の更新された要素だけを保持するために、疎データ構造を使用するという選択もある。

MPI を用いた実装では、MPI_REDUCE 関数をこの処理のために使用できる。【以上】

5.2 INDEPENDENT の進んだ例

```
      !HPF$ INDEPENDENT
      DO I = 1, 10
          WRITE (IOUNIT(I),100) A(I)
      END DO
100   FORMAT ( F10.4 )
```

IOUNIT(I) が、1 から 10 までのすべての I について異なる値に評価される場合、このループは繰返しごとに異なる入出力装置（したがって異なるファイル）へ書き込みを行う。その場合、このループは独立したものとして正しく記述される。一方、すべての I について IOUNIT(I)=5 の場合、この表明は誤りであり、ループは HPF 規格合致ではない。

```
!HPF$ INDEPENDENT, NEW (J)
      DO I = 2, 100, 2
      !HPF$ INDEPENDENT, NEW(VL, VR, UL, UR)
        DO J = 2 , 100, 2
          VL = P(I,J) - P(I-1,J)
          VR = P(I+1,J) - P(I,J)
          UL = P(I,J) - P(I,J-1)
          UR = P(I,J+1) - P(I,J)
          P(I,J) = F(I,J) + P(I,J) + 0.25 * (VR - VL + UR - UL)
        END DO
      END DO
```

J ループに NEW 節がなかったら、どのループも独立したものにならない。なぜなら、インターリーブにループの繰返しを実行することにより、ループの同じ繰返しの中で計算された値以外の VL, VR, UL,UR の値が P(I,J) への代入で使用される可能性があるからである。しかし、NEW 節は、ループの個々の繰返しで異なる記憶単位が使用される場合には、その可能性がないことを指定する。この実装を使用することにより、ループの繰返しが互いに独立したものになる。DO ループの刻み幅のため（つまり、I と J は常に偶数であり、したがって、I-1 などは常に奇数になるため）、配列 P の参照による干渉がないことに注意されたい。

ループが入れ子になっている場合、集計変数は、それに対する集計演算が内側ループの中にある場合でも、INDEPENDENT な外側ループで保護される必要があるかもしれない。さらにこのとき、その内側ループや中間のループは、INDEPENDENT であってもなくても構わない。

```
!     入れ子ループの例 1。内側 (INNER) ループが逐次。
```

```
      X = 10
OUTER: DO WHILE (X < 1000)   ! このループは逐次
         !HPF$ INDEPENDENT, NEW(J), REDUCTION(X)
MIDDLE:  DO I = 1, N
INNER:     DO J = 1, M
             X = X + J
             !  集計文を除いて、ここで X を参照すると正しくなくなることに
             !  注意されたい
           END DO INNER
         !  集計文を除いて、ここで X を参照すると正しくなくなることに
         !  注意されたい
         END DO MIDDLE
         PRINT *, X
       END DO OUTER
```

変数 X は、MIDDLE ループに対する REDUCTION 節に現われるので、そのループと、その内側のループを通じて、保護された集計変数となる。INNER に INDEPENDENT 指示文がある場合、その指示文の REDUCTION または NEW 節に X を含めるのは間違いである。最外側のループは INDEPENDENT でなく、それゆえ、X も MIDDLE ループの外側の範囲では保護する必要はなく、また保護することはできない。

NEW 節に現われた変数は、そのループとその外側のループで集計変数となることはできないが、その内側のループでは、集計変数として使用できる。

```
!    入れ子ループの例 2。外側 (OUTER) ループでの NEW 節。

!HPF$ INDEPENDENT, NEW(I)
OUTER:  DO K = 1, 100
          !HPF$ INDEPENDENT, NEW (J,X)
MIDDLE:   DO I = 1, N
            X = 10
            !HPF$ INDEPENDENT, REDUCTION(X)
INNER:       DO J = 1, M
               X = X + J**2
               !  集計文を除いて、ここで X を参照すると正しくなくなることに
               !  注意されたい
             END DO INNER
             Y(I) = X
           END DO MIDDLE
         END DO OUTER
```

ここでは、X は内側ループでのみ保護された集計変数である。

```
INTEGER, DIMENSION(M) :: VECTOR
```

```
!HPF$ INDEPENDENT, REDUCTION(X, Y)
      DO I = 1, N-4
        X(I:I+4) = X(I:I+4) + A(I)    ! 合わせて 5 回更新される
        Y(VECTOR) = Y(VECTOR) + B(I,1:N)
      END DO
```

もしこのループがブロックに分散されるのであれば、コンパイラは X の全体配列のプライベートな複製を作る必要がないことに注意されたい。

INDEPENDENT なループが活動状態である間に、集計文の形式をした文が現われたが、更新される変数は保護された集計変数ではないという場合、プログラマは、INDEPENDENT ループのどの二つの繰返しも、同じ位置を更新することがないということを保証していることになる。例えば、次の例である。

```
!HPF$ INDEPENDENT
      DO I = 1, N
        ! X はリダクション変数では「ない」が、私は INDX(1:N) に
        ! 同じ値が繰り返し現われることがないことを知っている。
        ! 更新は、X(INDX(I)) に対して直接行なってよい。
        X(INDX(I)) = X(INDX(I)) + F(I)
        ! 私は、IF 文の条件が I の高々一つの値に対して真で
        ! あることも保証する。
        IF (A(I) > B(I)) Y = Y + 1
      END DO
```