HPF/JA
言語仕様書

JAHPF (Japan Association for High Performance Fortran)

January 31, 1999
Version 1.0

本仕様書は、1996年7月から1997年1月までのJAHPF(Japan Association for High Performance Fortran)準備会および1997年1月から1999年1月までのJAHPFの活動によりまとめられたものである。また、本仕様書は、Rice大学が著作権を有するHPF 2.0 language specificationの拡張仕様を定めるものである。本仕様書に現れる、一部のHPF2.0仕様については、Rice大学の許可のもとで転載されたものである。

目 次

Acknowledgments

JAHPF(Japanese Association for High Performance Fortran) は、HPF の拡張仕様 HPF/JA を定めることにより HPF 言語の実用性を高めるとともに、HPF 言語の普及促進、HPF 利用技術の研究開発を行うことを目的とする任意団体である。本団体の活動は 1996 年 7 月の準備会結成に始まり、1997 年 1 月の本会議開始を経て、現在 (1999 年 1 月) に至っている。本ドキュメントにまとめられた HPF/JA 仕様は、JAHPF メンバの精力的な活動の結果である。JAHPF の現在のメンバは下記の通りである。

(敬称略、アルファベット順)

青山 裕司 ..... 東京大学
藤原 広行 ..... 防災科学技術研究所
柄谷 和輝 ..... 高度情報科学技術研究機構
林 康晴 ..... 日本電気
岩下 英俊 ..... 富士通
神谷 幸男 ..... 富士通
片山 博 ..... 日本電気
河合 伸一 ..... 地球シミュレータ研究開発センター
小林 篤 ..... 日立製作所
升本 順夫 ..... 東京大学
松元 亮治 ..... 千葉大学
松尾 裕一 ..... 航空宇宙技術研究所
三浦 謙一 ..... 富士通
三好 甫 ..... 地球シミュレータ研究開発センター
水見 俊介 ..... 日立製作所
長嶋 利夫 ..... 三菱総合研究所
中村 壽 ..... 高度情報科学技術研究機構
中村 孝 ..... 航空宇宙技術研究所
中尾 雅弘 ..... 三菱重工業
野方 康一 ..... 日立製作所
布広 永示 ..... 日立製作所
荻津 格 ..... イリノイ大学
太田 寛 ..... 日立製作所
大竹 和生 ..... 気象庁
岡部 寿男 ..... 京都大学
岡田 信 ..... 富士通
奥田 洋司 ..... 横浜国立大学

坂上 仁志 ..... 姫路工業大学
左近 彰一 ..... 日本電気
妹尾 義樹 ..... 日本電気
嶋 英志 ..... 川崎重工業
清水 鉄也 ..... 理化学研究所
新宮 哲 ..... 地球シミュレータ研究開発センター
新内 浩介 ..... 日立製作所
末広 謙二 ..... 日本電気
末安 直樹 ..... 富士通
鈴木 睦 ..... 宇宙開発事業団
高橋 正樹 ..... 地球シミュレータ研究開発センター
高橋 俊 ..... 日立製作所
田中 高史 ..... 郵政省 通信総合研究所
谷 啓二 ..... 地球シミュレータ研究開発センター
辻畑 好秀 ..... 日立製作所
山本 富士男 ..... 神奈川工科大学
山崎 昇 ..... 富士総合研究所
横川 三津夫 ..... 地球シミュレータ研究開発センター
渡辺 国彦 ..... 文部省 核融合研究所
(事務局)
秦 万美子 ..... (財) 高度情報科学技術研究機構
高橋 由香 ..... (財) 高度情報科学技術研究機構
以上 48 名

また、事務局を始め、委員旅費や会場の提供など JAHPF の種々の活動を援助していただいた (財) 高度情報科学技術研究機構 (RIST) に対して謝意を表する。

第1章　概要

本書は、HPF(High Performance Fortran) をより実用的にするために JAHPF(Japan Association for High Performance Fortran) で定められた HPF 拡張言語仕様 HPF/JA 1.0 を規定する。HPF/JA 1.0 は HPFF(High Performance Fortran Forum) によって定められた HPF 言語仕様への拡張および変更として設計されている。基準として使われている HPF 言語仕様は、現時点 (1999 年 1 月) では、HPF2.0 言語およびその公認拡張 (High Performance Fortran Language Specification Version 2.0, Jan. 31, 1997) である。

HPF/JA の拡張仕様の目的は、大きく分けると

- プログラムの並列処理記述性の向上、適用範囲の拡大
- ユーザによるきめ細かい並列化/最適化の記述を可能にすることにより、現段階のコンパイラ処理能力不足を補う

の 2 点である。

拡張機能は、大きく以下のように分類される。

1. 並列処理記述能力の拡大
   - REDUCTION 種別の指定
     … REDUCTION 節 (HPF 公認拡張機能) の適用範囲を拡大する。
2. 通信最適化
   - 非同期転送機能
     … プロセッサ間通信を計算とオーバラップさせる。
   - SHADOW 指示文の拡張
     … アクセスが高速であるフルシャドウ割付けを選択できる。
   - REFLECT 指示文
     … シャドウ領域の値を明示的に設定する。
   - ON 指示文の HOME 節の拡張
     … シャドウ領域を考慮した活動プロセッサを指定できる。
   - LOCAL 節／指示文の拡張
     … データのアクセスに通信が不要であることを指示する。
   - 通信スケジュール再利用
     … 同じパターンで繰返し行われる通信を効率化する。

第2章　記述法と構文

本章では、本書で使われている記述規約と HPF/JA 指示文の構文について記述する。

2.1　記述法

本書では HPF2.0 仕様および Fortran 95 規格と同じ記述法を使用する。特に、構文規則には同じ規約を使用する。言語機能の BNF 記述は HPF 仕様および Fortran 規格と同じ様式で与えられる。HPF/JA 構文規則と HPF 構文規則および Fortran 構文規則とを区別するために、各 HPF/JA 規則は J*snn* という形式の識別番号をもつ。ここで、*s* は章番号に対応しており、*nn* は 2 桁の順序番号である。本書で定義されてない非終端記号は HPF2.0 仕様または Fortran 規格の中で定義されている。H*snn* という形式の識別番号をもつ規則は HPF2.0 仕様の中で定義されており、R*snn* という形式の識別番号をもつ規則は Fortran 規格の中で定義されている。また、「マッピング」や「記憶単位」のようないくつかの技術用語は HPF2.0 仕様または Fortran 規格で定義されていることにも注意されたい。

HPF/JA の構文規則はしばしば HPF2.0 の類似の構文規則の拡張されたものである。このような場合には、非終端記号の名前は接尾辞 (suffix) *-ja* がついている。したがって、*name* または HPF 公認拡張仕様における *name-extended* のような非終端記号が再定義されたとき、構文規則の残りの中で、*name* または *name-extended* のすべての引用は *name-ja* で置き換えられるという条件の下で、それは *name-ja* として引用される。

> 【仕様の根拠】 本書の全体を通して、機能を含むこと、特定の機能定義を選択したこと、その他の決定を行ったことの根拠を説明する資料は、この形式で仕切られる。言語定義にのみ興味がある読者はこの部分を読み飛ばすことを望むかもしれない。一方、言語の設計に興味がある読者はこの部分をより注意深く読みたいかもしれない。【以上】

> 【利用者への助言】 本書の全体を通して、主に利用者のための資料 (構文と解釈の多くの例を含む) はこの形式で仕切られる。技術的な資料にのみ興味がある読者はこの部分を読み飛ばすことを望むかもしれない。一方、より教育的な方法を求める読者はこの部分をより注意深く読みたいかもしれない。【以上】

> 【実装者への助言】 本書の全体を通して、主に実装者のための資料はこの形式で仕切られる。言語定義にのみ興味がある読者はこの部分を読み飛ばすことを望むかもしれない。一方、コンパイラの実装に興味がある読者はこの部分をより注意深く読みたいかもしれない。【以上】

2.2　指示文の構文

HPF/JA 指示文は次の意味で HPF2.0 指示文および Fortran 構文と一貫性がある。すなわち、ある HPF/JA 指示文が将来の HPF 仕様の一部として採用された場合、HPF/JA プログラムを HPF プログラムに変換するために必要な変更は指示文先頭語 (hpfja-directive-origin) を `!HPF$`に置き換えるだけである。また、ある HPF/JA 指示文が将来の Fortran 規格の一部として採用された場合、HPF/JA プログラムを Fortran プログラムに変換するために必要な変更は指示文先頭語を空白に置き換えるだけである。

HPF/JA 指示文の一般的な形式を以下に示す。

J201　*hpfja-directive-line*　**is**　*hpfja-directive-origin*　*hpf-directive*

J202　*hpfja-directive-origin*　**is**　`!HPFJ`
or　`CHPFJ`
or　`*HPFJ`

次章以降で規定する HPF/JA 仕様を用いる場合には、指示文は *hpfja-directive-origin* で開始しなければならない。また HPF/JA 仕様を用いていない HPF2.0 の指示文も *hpfja-directive-origin* で開始してよい。HFP2.0 の指示文は、HPF2.0 の指示文先頭語 (directive-origin) で開始しても構わない。

> 【利用者への助言】 HPF2.0 処理系は持つが HPF/JA 処理系を持たないシステムを使用する可能性のある利用者は、HPF2.0 の指示文を`!HPF$`で開始する方がよい。そうすれば、HPF2.0 処理系を用いたとき少なくとも HPF2.0 指示文は有効になり、プログラムの可搬性が増すことになる。一方、HPF/JA 処理系しか用いない利用者は、`!HPFJ` のみを用いればよい。それにより、指示文が HPF2.0 仕様に含まれているかどうかを調べる面倒な作業から解放される。
>
> なお、HPF/JA 指示文は、それを無視したときに正しい HPF2.0 プログラムとなるように設計されている。【以上】

HPF/JA 指示文の文字種 (英大文字、英小文字)、行形式、空白、および継続行に関する規則は、HPF2.0 指示文の規則にしたがう。HPF/JA 指示行と HPF2.0 指示行は互いに継続行になってはならない。

次章以降では HPF2.0 で規定されている specification-directive-extended(H206)、executable-directive-extended(H207)、および executable-construct-extended(H208) に対して、いくつかの構文を追加および削除する。これらの変更をまとめて、新しい定義を以下に示す。

J203 *specification-directive-ja* **is** *processors-directive*
or *subset-directive*
or *align-directive*
or *distribute-directive*
or *inherit-directive*
or *template-directive*
or *combined-directive*
or *sequence-directive*
or *dynamic-directive*
or *shadow-directive*
or *asyncid-directive*

J204 *executable-directive-ja* **is** *independent-directive-ja*
or *realign-directive-ja*
or *redistribute-directive-ja*
or *on-directive*
or *resident-directive*
or *asynchronous-directive*
or *asyncwait-directive*
or *reflect-directive*
or *local-directive*
or *index-reuse-directive*

J205 *executable-construct-ja* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *if-construct*
or *where-construct*
or *on-construct*
or *resident-construct*
or *task-region-construct*
or *asynchronous-construct*
or *local-construct*

第3章　並列処理指定に関するHPF/JA拡張

3.1　REDUCTION 種別の指定

本拡張仕様の目的は、集計 (リダクション) 記述の自由度を増やすことである。

HPF2.0 の REDUCTION 節は、集計の種別を明示しない。代わりに、集計変数の参照形式を集計文 (*reduction-stmt*) の形に制限することにより、集計種別をコンパイラが判別することになっている（HPF2.0 仕様書 5.1.3 項）。このため、集計記述の自由度が制限されている。また、実用プログラムでよく用いられる MAXLOC，MINLOC 計算は、HPF2.0 の集計記述には含まれていない。したがってこれらを含むループに対して INDEPENDENT 指示を与えることはできない。

本拡張仕様は、REDUCTION 節に集計種別を明示し、集計変数を任意の形で参照できるようにするものである。また、MAXLOC，MINLOC 計算を含む INDEPENDENT ループを記述できるようにするものである。

3.1.1　構文

INDEPENDENT 指示文の構文規則 (HPF 仕様書 5.1 節の H501,H503) を、以下のように変更する。

J301　*independent-directive-ja*　**is**　INDEPENDENT [, *new-clause*]
　　[, *reduction-clause-ja-list*]

J302　*reduction-clause-ja*　**is**　REDUCTION
　　([*reduction-kind* :] *reduction-spec-list*)

J303　*reduction-kind*　**is**　*reduction-operator*
　　or　*reduction-function*
　　or　*maxmin-kind*

J304　*reduction-operator*　**is**　+
　　or　*
　　or　.AND.
　　or　.OR.
　　or　.EQV.
　　or　.NEQV.

J305　*maxmin-kind*　**is**　FIRSTMAX
　　or　FIRSTMIN
　　or　LASTMAX
　　or　LASTMIN

J306 *reduction-spec* **is** *reduction-variable* [**/** *location-variable-list* **/**]

J307 *location-variable* **is** *scalar-variable-name*

reduction-function は HPF 仕様書 5.1.3 項で定義されている。
HPF 仕様書 5.1 節にある制約に以下を追加する。

制約: *reduction-kind* が *maxmin-kind* であるとき、*reduction-spec* は *location-variable-list* を持たなければならない。*reduction-kind* が *maxmin-kind* でないとき、または *reduction-kind* が省略されたときは、*reduction-spec* は *location-variable-list* を持ってはならない。

制約: *reduction-kind* が *maxmin-kind* であるとき、*reduction-spec* 内の *reduction-variable* は *scalar-variable-name* でなければならない。

制約: *reduction-variable* に指定された変数の型は、*reduction-kind* のおのおのに対して以下のものでなければならない。

`.AND.`, `.OR.`, `.EQV.`, `NEQV.` に対しては論理型。

`IAND`, `IOR`, `IEOR` に対しては整数型。

`+`, `*`に対しては数値型。

`MAX`, `MIN`, `FIRSTMAX`, `FIRSTMIN`, `LASTMAX`, `LASTMIN` に対しては整数型か実数型。

制約: *reduction-kind* なしの *reduction-clause* 内で指定された *reduction-variable* は、ループ内で HPF2.0 仕様書 5.1.3 項に規定される集計文の形式で参照されなければならない。(*reduction-kind* 付きの *reduction-clause* 内で指定された *reduction-variable* は、ループ内で任意の形式で参照されてよい。)

また、HPF2.0 仕様書第 5.1 節の第 5 の制約は以下のように変更される。

制約: *reduction-variable* または *location-variable* として現れる変数は、同じ *independent-directive* の中で複数回現れてはならず、*independent-directive* が適用される後続の *do-stmt*、*forall-stmt* および *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* および *reduction-clause* に現れてはならない。

3.1.2 意味

INDEPENDENT 指示文は、`DO` ループの繰返しが互いに干渉しないことを表明する (HPF 仕様書 5.1 節)。集計種別付きの `REDUCTION` 節は、この干渉の条件を以下のように緩和する。

- 干渉の 1 番目の条件の 2 番目の例外を、以下のように変更する。下線部が変更点である。
  - 例外: 変数が<u>集計種別なしの</u>`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。

  また、以下の例外を追加する。

– 例外: 変数が集計種別付きの REDUCTION 節に集計変数または位置変数として現われる場合、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。ただし、その DO ループは、その変数に対応する集計種別と集計変数による集計演算を構成していなければならない。

- 干渉の 2 番目の条件の 2 番目の例外を、以下のように変更する。下線部が変更点である。
  – 例外: 変数が集計種別なしの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。

  また、以下の例外を追加する。
  – 例外: 変数が集計種別付きの REDUCTION 節に集計変数または位置変数として現われる場合、DO ループの異なる繰返しでのその変数に対する代入と使用は互いに干渉しない。ただし、その DO ループは、その変数に対応する集計種別と集計変数による集計演算を構成していなければならない。

ここで、**集計計算** (reduction computation) を構成するとは、以下のように定義される。ここでは、DO ループのある繰返しを一つのブロックと考えたとき、繰返し入口における変数 X の値を X^{in}、出口における値を X^{out} とする。X^{in} は仮想的な値であり、X が実際にそのような値を取り得るかどうかは問わない。X^{in} の値によっては、X^{out} は定義されない場合がある。

- R^{in} の値が仮に何であっても次式を常に成り立たせる、結合法則の成り立つ演算 f と、R^{in} に依存しない値 c が存在するとき、DO ループのその繰返しは変数 R に対して集計計算を構成するという。

$$R^{out} = f(R^{in}, c)$$

  このときの値 c を、集計要素と呼ぶ。また、f は表 3.1 で定義されるいずれかの演算でなければならない。
- DO ループのすべての繰返しが変数 R について同じ演算で集計計算を構成するとき、その DO ループは変数 R についてその演算で集計計算を構成すると言う。

なお、集計種別が “+” または “*” の場合には、実際の計算機では計算順序に依存する計算誤差があるため、R^{in} の値によって c の値がふらつくことがある。この点を考慮し、集計種別が “+” または “*” の場合には、上式の代りに以下の式を成り立たせる R^{in} に依存しない値の列 $c_1, c_2, \cdots, c_n\ (n \geq 0)$ が存在すれば、集計計算を構成していると考える。

$$R^{out} = f(\cdots f(f(R^{in}, c_1), c_2) \cdots, c_n)$$

3.1.3 制約

ここでは、集計計算の集計変数を R、位置変数を $L_1, \cdots, L_m\ (m \geq 0)$ とする。DO ループのある繰返しの終了時の値や状態が ${R^{in}}, {L_1}^{in}, \cdots, {L_m}^{in}$ の値によって変化しないとき、その値や状態は、その繰返しにおいてその集計計算に対して**不変である** (invariant) という。

ある値や状態が DO ループのすべての繰返しにおいて集計計算に対して不変であるとき、その値や状態は DO ループにおいて集計計算に対して不変であるという。

集計種別	$f(x,y)$
+	x + y
*	x * y
.AND.	x.AND.y
.OR.	x.OR.y
.EQV.	x.EQV.y
.NEQV.	x.NEQV.y
MAX	MAX(x, y)
MIN	MIN(x, y)
IAND	IAND(x, y)
IOR	IOR(x, y)
IEOR	IEOR(x, y)
FIRSTMAX	MAX(x, y)
FIRSTMIN	MIN(x, y)
LASTMAX	MAX(x, y)
LASTMIN	MIN(x, y)

表 3.1: 集計種別に対応する演算

1. 集計種別付きの REDUCTION 節を持つ INDEPENDENT 指定された DO ループは、その集計変数について集計計算を構成していなければならず、その集計種別と演算の対応は表 3.1 で許された組み合わせでなければならない。
2. 集計種別が *maxmin-kind* であるとき、その DO ループのすべての繰返しは、集計変数 R に対応するすべての位置変数 L_k について以下の条件を満たさなければならない。
   - R^{in} が R を更新する範囲にあるとき、${L_k}^{out}$ は必ず確定でなければならず、その値はその集計計算に対して不変でなければならない。
   - R^{in} が R を更新しない範囲にあるとき、${L_k}^{out}$ は ${L_k}^{in}$ が不定なら不定でなければならず、${L_k}^{in}$ が確定なら ${L_k}^{in}$ と同じ値でなければならない。

   ここで、R を更新する範囲、及び、更新しない範囲は、集計種別に応じて次の表で定義される。この表で c_i は、繰返し i における集計要素を表す。

集計種別	R を更新する範囲	R を更新しない範囲
FIRSTMAX	$R^{in} < c_i$	$R^{in} \geq c_i$
FIRSTMIN	$R^{in} > c_i$	$R^{in} \leq c_i$
LASTMAX	$R^{in} \leq c_i$	$R^{in} > c_i$
LASTMIN	$R^{in} \geq c_i$	$R^{in} < c_i$

3. 集計変数、位置変数、または NEW 変数を除くすべてのデータ実体の値と属性、およびファイルと装置の状態（有無、記録の内容、ファイル位置、その他 INQUIRE 文で問合わせることのできる性質）は、その DO ループが構成する集計計算のすべてに対して不変でなければならない。
4. 集計種別付き REDUCTION 節で指定された集計変数は、その変数によって構成される集計計算を除くすべての集計計算に対して不変でなければならない。

5. 集計種別付き REDUCTION 節で指定された位置変数は、同じ *reduction-spec* で指定された集計変数によって構成される集計計算を除くすべての集計計算に対して不変でなければならない。

【仕様の根拠】制約 1. の根拠

種別指定なしの集計変数は、集計文を使った場合に限ってアクセスが許される (HPF2.0 仕様書 5.1.3 項)。これに対して、種別指定付きの集計変数は、アクセスの方法は自由であるが、全体として集計計算を構成していなければならない。集計種別なし REDUCTION が構文で制約付けされるのに対し、集計種別付き REDUCTION はその計算の意味で制約付けされる。例えば sum であれば、どのような具体的な記述がされているかは問わず、足し加える計算 (の繰返し) を行っていることが条件となる。【以上】

【利用者への助言】 種別指定と集計計算の組み合わせに関するプログラムの間違いは、処理系で完全に検出することはできない。利用者は、集計計算の意味を十分に理解した上で、それを逸脱しないプログラミングを行う必要がある。【以上】

【例】 制約 1. の例

```
DO I=1,100
  X = X+A(I)
  IF (I.EQ.3) X = X+B
END DO
```

集計種別を “+” と考えると、各繰返しにおける集計要素 c は、以下のように X に依存しない式で得られる。

- I $\neq 3$ のとき、$X^{out} = X^{in} + \mathtt{A(I)}$
  すなわち、$c = \mathtt{A(I)}$
- I $= 3$ のとき、$X^{out} = X^{in} + \mathtt{A(3)} + \mathtt{B}$
  すなわち、$c = \mathtt{A(3)} + \mathtt{B}$
  (より厳密には、計算順序に依存する誤差を考えて、$c_1 = \mathtt{A(3)}, c_2 = \mathtt{B}$)

従って、この DO ループは REDUCTION 節

```
REDUCTION(+:X)
```

を記述するための制約 1. を満たす。

【例】 制約 2. の例

```
!HPFJ INDEPENDENT, REDUCTION(FIRSTMAX:AMAX/ILOC/)
      DO I=1,N
        IF (AMAX.LT.A(I)) THEN
          AMAX = A(I)
          ILOC = I
        END IF
      END DO
```

AMAX^{in} の値を変化させると、AMAX^{out} と ILOC^{out} は以下のように変化することが、プログラムから読み取れる。

AMAX^{in}	AMAX^{out}	ILOC^{out}
-HUGE	A(I)	I
⋮	A(I)	I
A(I)	A(I)=AMAX^{in}	ILOC^{in}
⋮	AMAX^{in}	ILOC^{in}
HUGE	AMAX^{in}	ILOC^{in}

この表より、集計要素 c は A(I) であることが分かる。そして、AMAX^{in} $<$ A(I) のとき ILOC^{out} $=$ I となり、AMAX^{in} $\geq$ A(I) のとき ILOC^{out} $=$ ILOC^{in} となることから、制約 2. を満たしていることが確認できる。

仮に、IF 文の条件節を (AMAX.LE.A(I)) に変更したとすると、表は次のように変る。

AMAX^{in}	AMAX^{out}	ILOC^{out}
-HUGE	A(I)	I
⋮	A(I)	I
A(I)	A(I)=AMAX^{in}	I
⋮	AMAX^{in}	ILOC^{in}
HUGE	AMAX^{in}	ILOC^{in}

この場合、AMAX^{in} $=$ A(I) のとき ILOC^{out} $=$ ILOC^{in} となると言えないことから、制約 2. を満たしていないと分かる。

3.1.4　記述例

```
!HPFJ INDEPENDENT, REDUCTION(MIN:AMIN), REDUCTION(+:S1,S2), NEW(TMP)
      DO I = 1,N
        IF(A(I).LT.AMIN) AMIN=A(I)
        TMP = S1+B(I)
        S1 = TMP+C(I)
        S2 = ADD(S2,D(I))
      END DO
```

ADD(x,y) は x + y を計算するだけのユーザ定義関数であるとする。
FIRSTMAX を使用する例を次に示す。

```
!HPFJ INDEPENDENT, NEW(I), REDUCTION(FIRSTMAX:AMAX/ILOC,JLOC/)
      DO J=1,N
        DO I=1,M
          IF(AMAX.LT.A(I,J)) THEN
            AMAX=A(I,J)
```

```
            ILOC=I
            JLOC=J
          END IF
        END DO
      END DO
```

第4章　通信最適化に関するHPF/JA拡張

4.1　非同期転送機能

プロセッサ間のデータ転送を、他の実行文の実行と並行して行うこと（非同期転送）を指定する。

データ転送の開始を指示する実行指示文と、終了を待ち合わせる実行指示文の組で表現する。これらの指示文は同じ ID を持たせることによって対応づける。

4.1.1　ASYNCID 宣言指示文

非同期転送の開始と終了の指示文を対応付けるための ID を宣言する。

4.1.1.1　書式

specification-directive-extended(H206) に *asyncid-directive* を追加する。

J401　*asyncid-directive*　　is　`ASYNCID` *async-id-list*

J402　*async-id*　　is　*async-id-name*

combined-attribute-extended(H801) に `ASYNCID` と `SAVE` を追加する。

制約: `SAVE` が *combined-directive* に現れるときには、必ず `ASYNCID` も現れなければならない。

【例】

```
ASYNCID ID1,ID2
ASYNCID :: X
ASYNCID,SAVE :: S,T,U
```

4.1.1.2　意味

ASYNCID 指示文　*async-id* が非同期識別子であることを宣言する。非同期識別子を使用する場合には、必ずこの宣言が必要である。

非同期識別子には以下のような性質がある。

- 類 (1) に属す局所要素 (local entity) である (JIS X 3001 プログラム言語 Fortran 14.1.2 項参照)。従って、その名前は有効域 (手続など) 内でのみ有効であり、同じ有効域内では類 (1) に属す他の局所要素[1]と同じ名前であってはならない。

[1]名前付き変数、文関数、組込み手続などがある。HPF ではこれにプロセッサとテンプレートが加わる。

- 参照結合 (モジュールで宣言して複数の有効域で参照すること) と親子結合 (親手続で宣言して親子の手続で共有すること) により、有効域を超えた結合ができる。
- 使用状態と不定状態のいずれかの状態を持つ。初期状態は不定状態であり、ASYNCHRONOUS 指示文 (4.1.2 項) で引用されると使用状態となり、ASYNCWAIT 指示文 (4.1.2 で引用されると再び不定状態となる。
- SAVE 属性を持つことができる。SAVE 属性を持つ非同期識別子は、RETURN 文または END 文の実行後も結合状態と割付け状態と使用／不定状態を保持する。

SAVE　非同期識別子が SAVE 属性を持つことを宣言する。

【仕様の根拠】 非同期識別子を新しい局所要素とした理由
非同期識別子を整数型の変数 (名前が意味をもつ) または整数式 (値が意味をもつ) とする案もあるが、以下の理由で新しい局所要素とする方がよいと考えた。

- 文法としてすっきりしている。
  - 非同期識別子として使用される名前であることが、ユーザにも処理系にも明確になる。
    プログラムの可読性が改善される。処理系でエラー検出できる機会が増える。処理系の最適化が促進される。
  - 識別子は指示文の中だけで出現する。
    指示文の追加だけで (Fortran 文の修正を行うことなく) プログラムの変更が可能になる。識別子に Fortran の変数や式を用いると、逐次解釈での翻訳時に宣言だけで使用されない変数を生じさせることがある。
- 処理系の実装が自然で容易。
  識別子の宣言をトリガにして実行時の構造を静的に準備できる。変数や式の値を識別子とすると、無駄な構造を生成したり、アーキテクチャに依存した実装上の困難が考えられる (例えば、基本整数型が 32 ビットでアドレス空間が 64 ビットの場合、基本整数型の値ではアドレスを保存するのに小さすぎるなど)。

【以上】

4.1.1.3　記述例

4.1.2.3 の記述例を参照。SAVE 宣言が必要な例は 4.1.5 項に示す。

4.1.2　ASYNCHRONOUS 指示文

ASYNCHRONOUS 指示文には単純指示文と指示構文があり、非同期転送の開始を指示する。ASYNCWAIT 指示文はその終了を待ち合わせる。

4.1.2.1　書式

executable-directive-extended(H207) に *asynchronous-directive* と *asyncwait-directive* を追加する。

executable-construct-extended(H208) に *asynchronous-construct* を追加する。

単純 ASYNCHRONOUS 指示文

J403 *asynchronous-directive* is ASYNCHRONOUS *asynchronous-stuff*

J404 *asynchronous-stuff* is ([ID =] *async-id*) [, *nobuffer-clause*]

【例】

```
ASYNCHRONOUS (ID=ID1)
ASYNCHRONOUS(ZZ)
```

ASYNCHRONOUS 指示構文

J405 *asynchronous-construct* is
hpfja-directive-origin block-asynchronous-directive
block
hpfja-directive-origin end-asynchronous-directive

J406 *block-asynchronous-directive* is ASYNCHRONOUS *asynchronous-stuff* BEGIN

J407 *end-asynchronous-directive* is END ASYNCHRONOUS

【例】

```
!HPFJ ASYNCHRONOUS(ID1) BEGIN
      A(:)=B(1:100)
      FORALL(I=1:M,J=1:N) S(I,J)=T(J,I)
!HPFJ END ASYNCHRONOUS
```

ASYNCWAIT 指示文

J408 *asyncwait-directive* is ASYNCWAIT ([ID =] *async-id*)

【例】

```
ASYNCWAIT(ID=ID1)
```

4.1.2.2 意味

以下の実行文と実行指示文を「非同期実行可能な文」と呼ぶ。

(1) 組込み代入文[2]
(2) body が組込み代入文である単純 FORALL 文
(3) REDISTRIBUTE 指示文

[2]通常の代入文。ユーザ定義代入文とポインタ代入文は含まれない。

(4) REALIGN 指示文
(5) REFLECT 指示文 (HPF/JA 拡張)

nobuffer-clause については 4.1.3 節を参照されたい。

単純 ASYNCHRONOUS 指示文　その直後に記述された非同期実行可能な文について、その実行によって生じるデータ転送の完了を待つことなく後続の処理を開始してよいことを指示する。

単純 ASYNCHRONOUS 指示文の実行により、*async-id* である転送識別子は使用状態となる。

ASYNCHRONOUS 指示構文　block に含まれるすべての非同期実行可能な文について、その実行によって生じるデータ転送の完了を待つことなく後続の処理を開始してよいことを指示する。

ASYNCHRONOUS 指示構文の実行により、*async-id* である転送識別子は使用状態となる。

ASYNCHRONOUS 指示構文は、以下のように複数の単純 ASYNCHRONOUS 指示文を用いた表現と等価である。

```
!HPFJ ASYNCHRONOUS(ID=id)
      BEGIN
      statement-1
      statement-2
      ...
      statement-n
!HPFJ END ASYNCHRONOUS
      ...
!HPFJ ASYNCWAIT(ID=id)
```

⇔

```
!HPFJ ASYNCHRONOUS(ID=id-1)
      statement-1
!HPFJ ASYNCHRONOUS(ID=id-2)
      statement-2
      ...
!HPFJ ASYNCHRONOUS(ID=id-n)
      statement-n
      ...
!HPFJ ASYNCWAIT(ID=id-1)
!HPFJ ASYNCWAIT(ID=id-2)
      ...
!HPFJ ASYNCWAIT(ID=id-n)
```

ASYNCWAIT 指示文　同じ *async-id* を持つ単純 ASYNCHRONOUS 指示文または ASYNCHRONOUS 指示構文によって開始された非同期転送の終了を待つことを指示する。

ASYNCWAIT 指示文の実行により、*async-id* である転送識別子は不定状態となる。

4.1.2.3　記述例

```
      REAL A(N),S(M,N),T(N,M)
!HPFJ ASYNCID ID1,ID2                          ! async-id
!HPF$ DISTRIBUTE A(BLOCK)
!HPF$ DISTRIBUTE (*,BLOCK) :: S,T
      ...
!HPFJ ASYNCHRONOUS (ID=ID1)
```

```
      FORALL(I=1,M) T(:,I)=A(:)*10.0          ! Tへの転送開始
      …                                       ! Tをアクセスしない処理
!HPFJ ASYNCWAIT (ID=ID1)                      ! Tへの転送終了
!HPFJ ASYNCHRONOUS (ID2) BEGIN
!HPF$ REDISTRIBUTE A(BLOCK)
      FORALL(I=1:M,J=1,N) S(I,J)=T(J,I)
!HPFJ END ASYNCHRONOUS                        ! A,Sへの転送開始
      …                                       ! A,Sをアクセスしない処理
!HPFJ ASYNCWAIT(ID2)                          ! A,Sへの転送終了
      …
```

4.1.2.4 制約

基本的な制約

1. 単純 ASYNCHRONOUS 指示文及び ASYNCHRONOUS 指示構文の対象となる実行文と実行指示文は、非同期実行可能な文 (4.1.2.2 節参照) でなければならない。
2. 単純 ASYNCHRONOUS 指示文または ASYNCHRONOUS 指示構文の実行時には、転送識別子は使用状態であってはならない。ASYNCWAIT 指示文の実行時には、転送識別子は使用状態でなければならない。

対象変数に関する制約 ASYNCHRONOUS 指示文の対象となる実行文と実行指示文について、それぞれ以下のものを非同期転送の対象変数[3]と呼ぶ。

対象となる文	対象変数
組込み代入文	左辺
単純 FORALL 文	body の代入文の左辺
REDISTRIBUTE 指示文	再分散の対象 (distributee) と、それに最終的に整列しているすべてのデータ実体
REALIGN 指示文	再整列の対象 (alignee)
REFLECT 指示文	REFLECT 転送の対象

1. ASYNCHRONOUS 指示文が実行されてから対応する ASYNCWAIT 指示文が実行されるまでの間に実行される文は、対象変数の引用を含んでいてはならない。ただし、以下の引用は許される。
   - 対象変数の属性 (型、形状、割付け状態など) を問い合わせるための引用。
   - マッピングを参照するための引用 (ON 指示文の HOME 節での引用など)。ただし、REDISTRIBUTE 指示文と REALIGN 指示文の対象変数については許されない。

【例】

```
      REAL A(M,N),B(M,N)
!HPFJ ASYNCID :: ID1
```

[3] ここで言う「変数」とは Fortran 90 仕様で言う変数なので、名前付き変数 (全体配列) だけでなく部分配列、配列要素、構造体の部分実体、文字部分列なども含む。

```
!HPF$ DISTRIBUTE B(BLOCK,*)
!HPF$ ALIGN A(:,:) WITH B(:,:)
!HPF$ DYNAMIC A,B
      …
!HPFJ ASYNCHRONOUS(ID=ID1)
!HPF$ REDISTRIBUTE B(*,BLOCK)     ! 対象変数は A と B
      …
      C(I)=A(I)                   ! 禁止：A は参照できない
      B=D+E                       ! 禁止：B は定義できない
      CALL SUB(B)                 ! 禁止：B は実引数として引用できない
      DEALLOCATE(A)               ! 禁止：A は不定にできない
!HPF$ REALIGN A(:,:) WITH T(:,:)  ! 禁止：A は再配置できない
      …
!HPFJ ASYNCWAIT(ID=ID1)
```

【仕様の根拠】 実引数としての引用を禁止する理由
実引数の値が非同期転送で定義された後で、仮引数の値によって書きつぶされるおそれがあるため。
コンパイラは引数渡しを値結合 (サブプログラム入口で実引数から局所変数にコピーし、出口で局所変数から元の実引数へコピーする方法) で実現するかもしれないし、サブプログラムの入口と出口で自動的な再分散を行うかもしれない。【以上】

2. ASYNCHRONOUS 指示構文では、対象変数として出現した変数は、構文内でその非同期実行可能な文より後で再び引用されてはならない。

【例】　下線の付いた変数は非同期転送の対象変数。

```
!HPFJ ASYNCHRONOUS(ID=ND) BEGIN
      A(1:N)=B(1:N)
      C(:)=A(:)+D(:)              !(a) 不可
      P(:)=D(:)                   !(b) 可
!HPF$ REALIGN B(:) WITH T(:)      !(c) 可
      A(N+1:NN)=E(N+1:NN)         !(d) 可
      FORALL(I=1:9) G(I+1)=G(I)   !(e) 可
!HPFJ END ASYNCHRONOUS
```

(a)　A の部分配列は対象変数なので参照できない。
(b)　D は複数出現するが、対象変数として出現していないので許される。
(c)　B は対象変数としては初めての出現なので許される。
(d)　A の部分配列は対象変数だが、重なりがないので許される。
(e)　対象変数が同一の文で参照されることは許される。

非同期再整列に固有の制約　対象が REALIGN 指示文である場合、以下の制約がある。

1. ASYNCHRONOUS 指示文によって非同期転送が開始されてから対応する ASYNCWAIT 指示文が実行されるまでの間、REALIGN 指示文の最終的 (ultimately aligned) な整列先 (下記の例では変数 C) に対して、直接または間接に以下のことを行ってはならない。
   - 割付けの開放、割付けを不定にすること
   - 手続呼出しの実引数としての引用
   - 再マッピング (非同期を含む)

   【例】

```
      !HPFJ ASYNCID ID1                          ! async-id
            REAL A(100,200)
            REAL B1(100,200),C1(100)
      !HPF$ DISTRIBUTE C1(BLOCK)
      !HPF$ ALIGN B1(:,*) WITH C1(:)
            REAL B2(100,200),C2(200)
      !HPF$ DISTRIBUTE C2(BLOCK)
      !HPF$ ALIGN B2(*,:) WITH C2(:)
      !HPF$ ALIGN A(:,:) WITH B1(:,:)            ! A は最初 B1 に整列
      !HPF$ DYNAMIC A,B1,B2,C1,C2
            …
      !HPFJ ASYNCHRONOUS(ID=ID1)
      !HPF$ REALIGN A(:,:) WITH B2(:,:)          ! A の非同期再整列開始
            …
      !HPF$ REDISTRIBUTE C2(BLOCK,:)             ! 禁止
            CALL FOO(C2)                         ! 禁止
            DEALLOCATE C2                        ! 禁止
            …
      !HPFJ ASYNCWAIT(ID=ID1)                    ! A の非同期再整列終了
            …
```

活動プロセッサの制約　対応する非同期転送指示文と転送待合せ指示文は、同じ活動プロセッサの集合で実行されなければならない。

【例】

```
!HPF$ ON (P(1:4)) BEGIN              ! 活動プロセッサは P(1:4)
!HPFJ ASYNCHRONOUS(ID=ID1)
      …
!HPFJ ASYNCHRONOUS(ID=ID2)
      …
!HPFJ ASYNCHRONOUS(ID=ID3)
      …
!HPF$ END ON
```

```
!HPFJ ASYNCWAIT(ID=ID1)            ! 不可。活動プロセッサは全プロセッサ

!HPF$ ON (P(5:8)) BEGIN
!HPFJ ASYNCWAIT(ID=ID2)            ! 不可。活動プロセッサは P(5:8)
!HPF$ END ON

!HPF$ ON (P(1:4)) BEGIN
!HPFJ ASYNCWAIT(ID=ID3)            ! 可。活動プロセッサは P(1:4)
!HPF$ END ON
```

4.1.3　ASYNCHRONOUS 指示文の NOBUFFER 節

代入文と FORALL 文に対する非同期転送をより効率的に行うための記述方法として、NOBUFFER 節を提供する。

4.1.3.1　書式

asynchronous-directive と *block-asynchronous-directive* (4.1.2.1 節) に NOBUFFER 節をオプショナルに記述できる。

J409　*nobuffer-clause*　　is　NOBUFFER

【例】

```
ASYNCHRONOUS (ID=ID1), NOBUFFER
ASYNCHRONOUS(ZZ),NOBUFFER
```

【例】

```
!HPFJ ASYNCHRONOUS(ID=Z), NOBUFFER BEGIN
      A(:)=B(:)
      FORALL(I=1:N) S(:,I)=T(I,:)
!HPFJ END ASYNCHRONOUS
```

4.1.3.2　意味

以下の実行文を「バッファなし非同期実行可能な文」と呼ぶ。

(1) 右辺が一つの変数 (全体配列、部分配列、配列要素、またはスカラ変数) だけから成る代入文
(2) (1) を body の代入文とする FORALL 文

NOBUFFER 節は、バッファなし非同期実行可能な文の右辺について、ASYNCHRONOUS 指示文によって非同期転送が開始されてから対応する ASYNCWAIT 指示文が実行されるまでの間、直接または間接に以下のことを行わないことを宣言する。

- 値の定義、値を不定にすること (値の参照は許される)
- 割付けの開放、割付けを不定にすること
- 手続呼出しの実引数としての引用
- 再マッピング (非同期を含む)
- マッピングを参照するための引用 (ON 指示文の HOME 節など)

【仕様の根拠】 NOBUFFER 節はバッファを使用しない非同期転送をコンパイラに強制するものではなく、バッファを使用しない非同期転送が可能である条件がそろっていることをコンパイラに教えるためのものである。【以上】

【実装者への助言】 NOBUFFER 節を持つ ASYNCHRONOUS 指示文は、バッファを経由しない転送で実現する方が効率がよい場合にはそうすることが望ましいが、それを強制するものではない。記述された代入文の種類やアーキテクチャに合わせて効率のよい方法を選択すればよい。【以上】

4.1.3.3 記述例

```
      REAL A(1000),B(1000)
      REAL C(100,100),D(100,100)
      INTEGER E(200),F(100,200,300)
      REAL S(500,20),T(800,20)
      INTEGER IX1(N),IX2(N)
!HPFJ ASYNCID :: DD
      …
!HPFJ ASYNCHRONOUS(ID=DD), NOBUFFER BEGIN
      A=B                                  ! 全体配列から全体配列への転送
      E=F(J,:,K+1)                         ! 部分配列から全体配列への転送
      FORALL(I=1,N) C(:,I)=D(I,:)          ! 部分配列間の transpose 転送
      S(IX1,:)=T(IX2,:)                    ! ベクトル添字付きの転送
!HPFJ END ASYNCHRONOUS
      …                  ! ここには、A,E,C,S についてアクセスがなく、
      …                  ! B,F,D,T について値の参照以外のアクセスがない。
!HPFJ ASYNCWAIT(DD)
```

4.1.3.4 制約

1. NOBUFFER 節を持つ単純 ASYNCHRONOUS 指示文と ASYNCHRONOUS 指示構文の対象となる実行文と実行指示文は、バッファなし非同期実行可能な文 (3.2 節参照) でなければならない。

2. NOBUFFER 節を持つ ASYNCHRONOUS 指示構文では、バッファなし非同期実行可能な文の右辺として出現した変数は、構文内で転送の対象変数として出現してはならない。

【例】　下線の付いた変数はバッファなし非同期実行可能な文の右辺。

```
!HPFJ ASYNCHRONOUS(ID=ND), NOBUFFER BEGIN
      A(1:N)=B(1:N)
      B(:)=C(:)                          !(a) 不可
      D(:)=C(:)                          !(b) 可
      FORALL(I=1:9) G(I+1)=G(I)          !(c) 不可
      S(1:100)=T(1:100)
      T(101:200)=U(1:100)                !(d) 可
!HPFJ END ASYNCHRONOUS
```

(a)　B(1:N) の範囲は右辺で参照されている。
(b)　C(:) は複数出現するが、対象変数として出現していないので許される。
(c)　G に重なりがある。同一の文でも重なりは許されない。
(d)　T の要素は対象変数と右辺で重なりがない。

4.1.4　ASYNC 接頭辞

REDISTRIBUTE 指示文、REALIGN 指示文、及び REFLECT 指示文の非同期実行は ASYNCHRONOUS 指示文と組み合わせることによって記述できるが、表現をより簡便にするため、ASYNC 接頭辞 (prefix) を提供する。

4.1.4.1　書式

redistribute-directive(H802) と *realign-directive*(H803) を以下のように変更する。

J410　*redistribute-directive-ja*　**is**　[*async-prefix*] *redistribute-directive*

J411　*realign-directive-ja*　**is**　[*async-prefix*] *realign-directive*

J412　*async-prefix*　**is**　ASYNC ([ID =] *async-id*)

【仕様の根拠】 *reflect-directive* の書式は 4.4 節で定義する。【以上】

【例】

```
ASYNC(ID=Z) REDISTRIBUTE D(BLOCK,*) ONTO PROC
ASYNC (ID) REDISTRIBUTE (CYCLIC) ONTO P :: T1,T2
ASYNC(ID2) REALIGN A(:,:) WITH B(:,:)
ASYNC(ID=Y) REALIGN (*,I) WITH T(I+1) :: A,B,C
ASYNC(ID=MM) REFLECT A
```

4.1.4.2 意味

async-prefix を持つ実行指示文（REDISTRIBUTE 指示文、REALIGN 指示文、及び REFLECT 指示文に限る）は、次のように ASYNCHRONOUS 指示文と組み合わせた記述と等価である。

`!HPFJ ASYNC(ID=`*id*`)` *executable-directive*	⇔	`!HPFJ ASYNCHRONOUS(ID=`*id*`)` `!HPFJ` *executable-directive*

4.1.4.3 記述例

```
!HPFJ ASYNCID ID1                              ! async-id
      REAL A(100,100),D(100,100)
!HPF$ ALIGN A(I,J) WITH D(I,J)
!HPF$ DISTRIBUTE D(*,BLOCK)
!HPF$ DYNAMIC A,D
      …
!HPFJ ASYNC(ID1) REDISTRIBUTE D(BLOCK,*)  ! A と D の再配置開始
      …                                    ! A と D をアクセスしない処理
!HPFJ ASYNCWAIT(ID1)                       ! A と D の再配置完了
      …                                ! A と D は新しいマッピングでアクセス可能
```

4.1.5 有効域に関する注意事項

ASYNCHRONOUS 指示文と ASYNCWAIT 指示文の制約は、4.1.2.4 項に述べた通りである。この節では、その制約を満たすためにプログラミングにおいて注意すべきことを解説する。

4.1.5.1 有効域をまたぐ非同期転送

ASYNCHRONOUS 指示文と ASYNCWAIT 指示文が異なる有効域 (scoping unit) にある場合、ASYNCWAIT 指示文が実行されるまで、非同期識別子と対象変数の割付けが不定にならないように注意しなければならない。そのためには、非同期識別子と対象変数を以下のいずれかの方法で大域的に宣言する方法を推奨する。

- それらの有効域が共通に参照するモジュールで宣言する。
- それらの有効域が親手続と内部手続の関係にある場合、または共通な親手続を持つ内部手続同士の関係にある場合、親手続で宣言する。

【例】 モジュールを使って、手続をまたぐ非同期転送を記述した例。

- モジュール

```
      MODULE MOO
!HPFJ ASYNCID Z
      REAL A(100),D(100)
!HPF$ ALIGN A(:) WITH D(:)
```

```
!HPF$ DISTRIBUTE D(BLOCK)
!HPF$ DYNAMIC A,D
      END
```

- 呼出し側

```
      PROGRAM MAIN
      USE MOO
      …
      CALL ASYNC_SUB
      …
      CALL ASYNCWAIT_SUB
      …
      END
```

- 転送を開始するサブルーチン

```
      SUBROUTINE ASYNC_SUB
      USE MOO
!HPFJ ASYNC(Z) REDISTRIBUTE D(CYCLIC)
      END SUBROUTINE
```

- 転送を待ち合わせるサブルーチン

```
      SUBROUTINE ASYNCWAIT_SUB
      USE MOO
!HPFJ ASYNCWAIT(Z)
      END SUBROUTINE
```

【例】　同じ内容を、親子結合を使用して記述した例。

```
      PROGRAM MAIN
!HPFJ ASYNCID Z
      REAL A(100),D(100)
!HPF$ ALIGN A(:) WITH D(:)
!HPF$ DISTRIBUTE D(BLOCK)
!HPF$ DYNAMIC A,D
      …
      CALL ASYNC_SUB
      …
      CALL ASYNCWAIT_SUB
      …
      CONTAINS
      SUBROUTINE ASYNC_SUB
!HPFJ ASYNC(Z) REDISTRIBUTE D(CYCLIC)
```

```
      END SUBROUTINE
      SUBROUTINE ASYNCWAIT_SUB
!HPFJ ASYNCWAIT(Z)
      END SUBROUTINE
      END
```

転送識別子と転送対象変数は、手続間を引数結合を通して受け渡すことはできない。

【仕様の根拠】 転送識別子はデータ実体ではないので、手続間で引数による受け渡しはできない。また、転送対象変数については、実引数として引用することが禁止されている (4.1.2.4 項)。

仮引数は手続の実行を終了すると不定になる。従って、仮引数を対象変数とする非同期転送は、必ず手続内で待合せを行わなければならない。仮引数と実引数が物理的に同じ領域にある保証はないので、手続から復帰した後で実引数への転送を待ち合わせるようなプログラムを書くことはできない。【以上】

【例】 誤ったプログラムの例

- モジュール

```
      MODULE MOO
!HPFJ ASYNCID ID1
      REAL B(50,100)
      END
```

- 呼出し側

```
      USE MOO
      REAL A(100,100)
      …
      CALL FOO(A(1:50,:))
!HPFJ ASYNCWAIT (ID1)        ! A への非同期転送を待ち合わせたい。
      …
```

- サブルーチン

```
      SUBROUTINE FOO(X)
      REAL X(50,100)
      …
!HPFJ ASYNCHRONOUS (ID1)
      X=B
      RETURN              ! 禁止: 仮引数 X の割付けはここで不定になる。
      END
```

4.1.5.2　同一サブプログラムの異なる呼出し間の非同期転送

ASYNCHRONOUS 指示文と ASYNCWAIT 指示文が同一のサブプログラムにある場合でも、同一のインスタンスで実行されない場合 (例えば、最初の呼出しで非同期転送を開始し、次の呼出しで終了待ちするような場合)、非同期転送中に転送識別子と対象変数の割付けが不定にならないように注意しなければならない。そのためには、4.1.5.1 項で述べたように転送識別子と対象変数を大域的に宣言するか、あるいは SAVE 属性付きで宣言することを推奨する。

【例】　以下のような場合、対象変数 A と転送識別子 ID はサブルーチン終了後も不定になってはならないので、SAVE 宣言を使用する。

呼出し側

サブルーチンを N 回呼び出す。

```
      DO I=1,N
        CALL PIPELINETRANS(I,N)
        …
      END
```

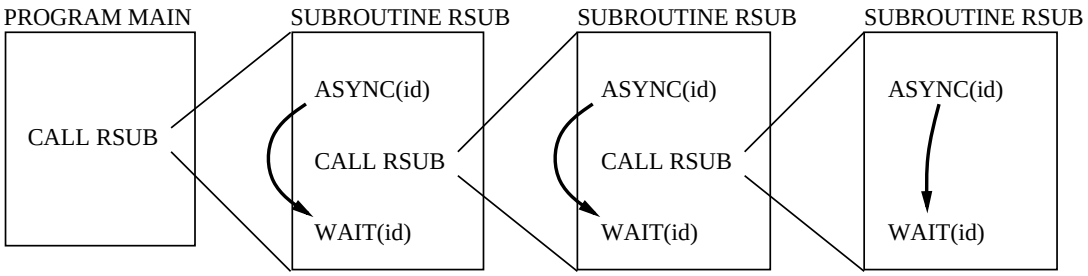
サブルーチン

変数 A から変数 ATMP を計算し、次回呼出しまでに非同期転送で ATMP から A へ値を戻す。

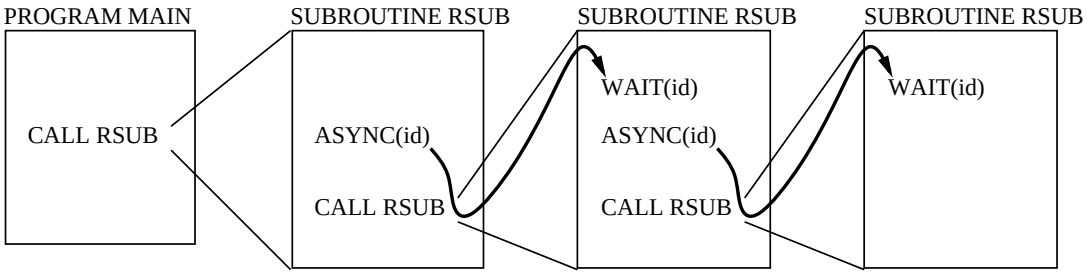
```
      SUBROUTINE PIPELINETRANS(NTIMES,NEND)
      REAL A(1000),ATMP(1000)
      SAVE A
!HPFJ ASYNCID,SAVE :: Z
! ---------------------------  ! ２回目以降の呼出しでは待ち合わせる。
      IF(NTIMES>1) THEN
!HPFJ   ASYNCWAIT(Z)
      END IF
! ---------------------------  ! A から ATMP を計算する。
      DO I=2,999
        ATMP(I)=0.25*(A(I-1)+2*A(I)+A(I+1))
      END DO
! ---------------------------  ! 最後以外の呼出しでは転送開始する。
      IF(NTIMES<NEND) THEN
!HPFJ   ASYNCHRONOUS(Z)
        A(2:999)=ATMP(2:999)
      END IF
! ---------------------------  ! 非同期転送を実行しながら呼出しに復帰。

      RETURN
      END
```

4.1.5.3 再帰手続内の非同期転送

再帰手続内で行われる非同期転送には、図 4.1 の (a) のように各インスタンス内で行う場合と、(b) のようにインスタンスをまたいで行う場合が考えられる。前者の場合には、非同期識別子と対象変数は手続内で宣言し、かつ SAVE 属性を付けないようにすればよい。後者の場合には、非同期識別子と対象変数に SAVE 属性を付けて宣言するか、モジュールで宣言するなど大域的な宣言となるようにすればよい。



(a) 各インスタンスに閉じた非同期転送



(b) インスタンスをまたぐ非同期転送

図 4.1: 再帰手続での非同期転送

4.1.5.4 非同期再マッピングに関する注意

非同期再分散では、再分散される変数 (distributee) だけでなく、それに整列している変数も転送対象変数となる。転送対象変数のすべてが非同期転送中に不定にならないように注意しなければならない。

非同期再整列では、最終的な整列先が非同期転送中に不定にならないように注意しなければならない。

【例】 誤ったプログラムの例

サブルーチン内で D に整列した変数 A が非同期転送中に不定となるため、処理系は動作を保証できない。A に関する宣言をモジュール内に移せば正しいプログラムとなる。

- モジュール

```
MODULE MODD
```

```
      REAL D(1000)
!HPF$ DISTRIBUTE(BLOCK),DYNAMIC :: D
!HPFJ ASYNCID :: ZZ
      END MODULE
```

- 呼出し側

```
      USE MODD
      …
      CALL MISDIST
!HPFJ ASYNCWAIT(ID=ZZ)              ! D の再分散を待ち合わせる。
      …
```

- サブルーチン

```
      SUBROUTINE MISDIST
      USE MODD
      REAL A(1000)
!HPF$ ALIGN(:) WITH D(:), DYNAMIC :: A     ! A を D に整列させる。
      …
!HPFJ ASYNCHRONOUS(ID=ZZ)
!HPF$ REDISTRIBUTE(CYCLIC) :: D      ! A と D が対象変数。
      RETURN                         ! 禁止:局所変数 A が不定になる。
      END
```

4.2 SHADOW 指示文の拡張

本節では、SHADOW 指示文の拡張に関して述べる。

例えば、以下のようなマッピング指示文が指定された場合、

【例】

```
      REAL A(4,4)
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO P
```

多くの HPF 処理系は、宣言された配列全体の領域のうち、各プロセッサ上に、ローカルな部分だけを割り付ける (図 4.2 参照)。

これに対して、*shadow-target* の各次元に、“*” を指定できるよう、SHADOW 指示文を拡張することにより、宣言された配列全体の領域を、各プロセッサへ割り付けるよう指示することが可能になる (図 4.3 参照)。

【例】

```
      REAL A(4,4)
```

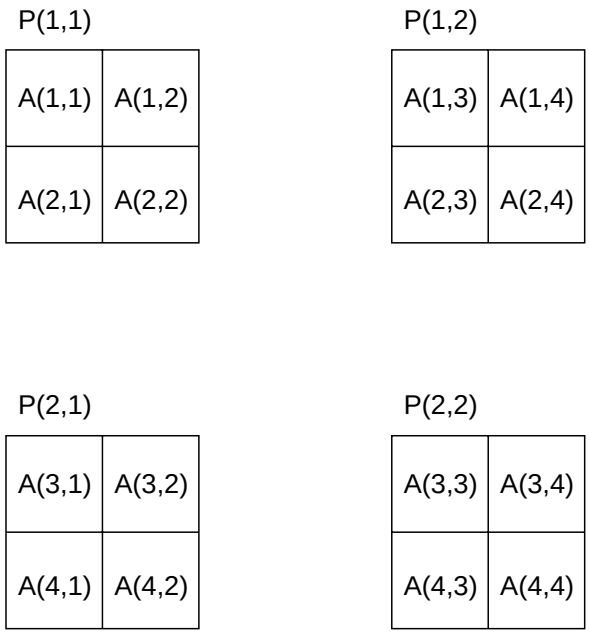


図 4.2: 通常の割付け方式

```
!HPF$ PROCESSORS P(2,2)
!HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO P
!HPFJ SHADOW A(*,*)                          ! SHADOW 指示文の拡張
```

この形式の SHADOW 指示文が指定された実体の SHADOW 属性を、特に「フル SHADOW 属性」と呼ぶことにする。

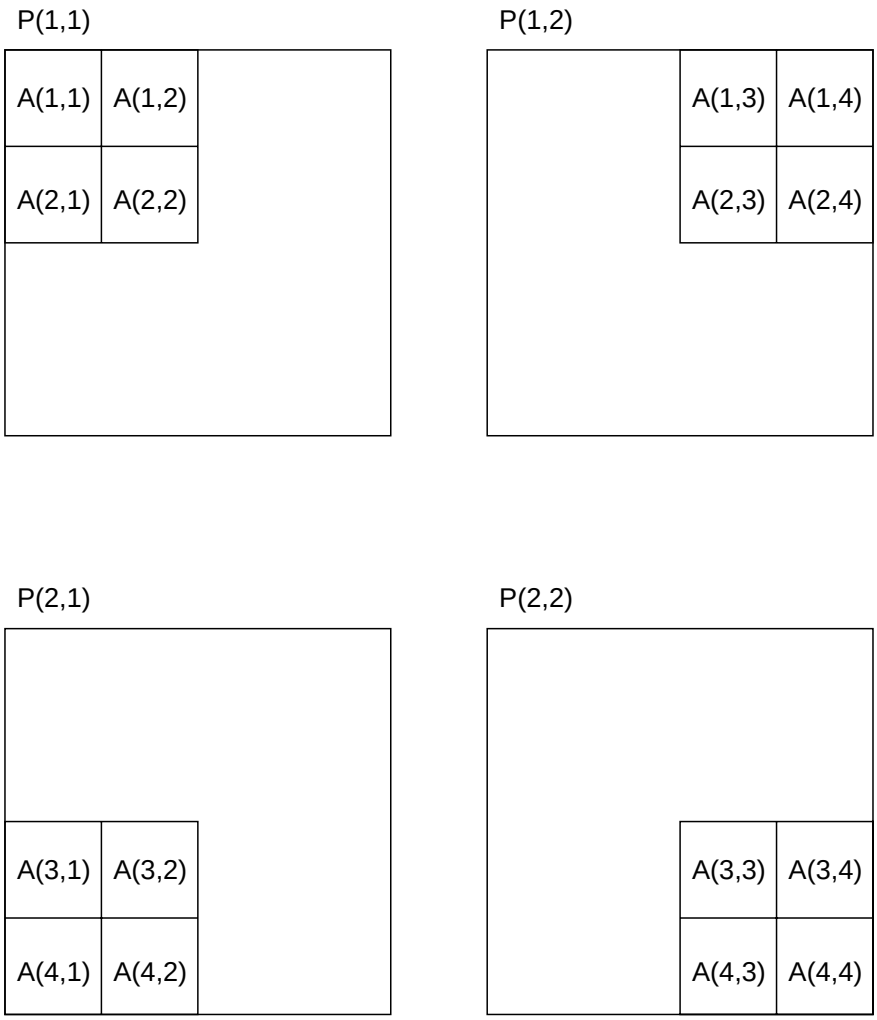


図 4.3: フル SHADOW 属性を持つ場合の割付け方式

このような割付け方式は、

- メモリの利用効率が悪い

ため、利用できるデータの大きさに対する制限が強くなる、という問題がある半面、

- 言語処理系は、配列がフル SHADOW 属性を持つと静的に判定できれば、その参照時に、グローバル添字からローカル添字への変換などの添字変換を行う必要がない。
- 配列の (明示的あるいは暗黙的な) 動的再マッピングの際、新たに領域を確保したり、元の領域から値をコピーしたりする必要がない。

という特徴により、目的プログラムの実行が高速に行える、という利点がある。
したがって、本指示文は、以下のような目的で用いることができる。

- アプリケーションプログラムの開発段階で、高速に実行テストを行う。
- メモリの許す限り性能を追求することにより、実行速度と問題の大きさとの間で、バラ

ンスの取れたプログラムを開発する。

【利用者への助言】 INHERIT 指示文を指定した仮引数に、さらに SHADOW 指示文を指定することはできない。(High Performance Fortran Language Specification Version 2.0 の 4.4.2 項、8.1 節、及び 8.13 節を参照されたい。)

したがって、INHERIT 属性を持つ仮引数が、フル SHADOW 属性を持つ実引数と結合する場合、言語処理系は、その仮引数の割付けがフル SHADOW 属性を持つ実体と同等であることを、静的に判定することができない可能性があり、その結果、上に述べたような、言語処理系は添字変換を行う必要がない、というフル SHADOW 属性の長所は失われる可能性が高い。そのため、必ずしも目的プログラムの実行は高速にならない場合がある。

したがって、利用者には、INHERIT 属性を持つ実体を、フル SHADOW 属性を持つ実体と結合させないよう強く推奨する。【以上】

【利用者への助言】 フル SHADOW 属性を持つ実引数とフル SHADOW 属性を持たない仮引数の結合、又はフル SHADOW 属性を持たない実引数とフル SHADOW 属性を持つ仮引数の結合を行うと、SHADOW 属性の変換のための実行時オーバヘッドが必要になる。

さらに、各プロセッサ上に、ローカルな部分だけを割り付ける場合の、割付けメモリ量が少なくてすむ、という長所を損なう可能性もある。

したがって、利用者には、この様な結合を行わないよう強く推奨する。【以上】

【実装者への助言】 本節の拡張は、フル SHADOW 属性を持たない実体に対して、実装者が図 4.3 のような割付け方式を採用することを妨げるものではない。【以上】

4.2.1 構文

SHADOW 指示文の構文規則中の *shadow-spec*(H820) を、以下のように拡張する。

J413 *shadow-spec-ja* **is** *width*
or *low-width* : *high-width*
or *full-width*

J414 *full-width* **is** *

【例】

```
      REAL A(100,100,100),B(100,100,100)
!HPF$ SHADOW A(5:5,0:1,3)                ! 従来の SHADOW 指示文
!HPFJ SHADOW B(*,*,*)                    ! SHADOW 指示文の拡張
```

4.2.2　制約

制約: *shadow-spec-ja-list* の長さは、*shadow-target* の次元数と等しくなければならない。

制約: *shadow-spec-ja* として *full-width* を指定した場合、すべての次元で *full-width* を指定しなければならない。

4.2.3　拡張された SHADOW 属性の同値関係

ここでは、拡張された SHADOW 指示文に対して、SHADOW 属性の同値関係を以下のように定義する。(High Performance Fortran Language Specification Version 2.0 の 8.13 節を参照されたい。)

1. *shadow-spec-ja* が *full-width* の場合、それは *full-width* であるような *shadow-spec-ja* とだけ同値である。
2. *shadow-spec-ja* が *full-width* でない場合、その中の式 w_1 と w_2 は、それらが同じ値を持つ場合、かつその場合に限り同値である。
3. *shadow-spec-ja* w は、*shadow-spec-ja* w*:*w と同値である。
4. *shadow-spec-ja* l_1*:*h_1 と *shadow-spec-ja* l_2*:*h_2 は、l_1 が l_2 と同値であり、かつ h_1 が h_2 と同値である場合、かつその場合に限り同値である。
5. 上に述べた以外の字面上異なる *shadow-spec-ja* 指定は、同値ではない。

ここで、2 つの SHADOW 属性は、一方の各 *shadow-spec-ja* が、他方の対応する *shadow-spec-ja* と同値である場合、かつその場合に限り同値である。

これにより、フル SHADOW 属性を持つ実体のマッピングを含むマッピングの集合に対して、同値関係が定義される。

4.3　明示的シャドウ

公認拡張機能のシャドウはコンパイラの最適化を促進することが目的であり、どのような最適化が行われるかはコンパイラに依存するため、最適化を補助するような一般的な指示文は導入しにくいと考えられる。しかし、最適化だけでは十分な性能が得られない場合も多くある。そこで、コンパイラの最適化を補助するという考え方でなく、シャドウ領域のアクセスを直接に指示するという考え方に基づく仕様を提案する。この機能は、コンパイラの最適化に依存しないユーザ記述による性能向上を狙うが、同時にコンパイラによる最適化を阻害しないことも目指している。

4.3.1　用語

Fortran では、変数、定数、及び定数の部分実体をまとめてデータ実体 (data object) と呼ぶ。データ実体は Fortran で規約された方法で値の定義及び参照ができる。これに対し、SHADOW 指示文 (公認拡張仕様) によって宣言された記憶領域に保持されるデータを、**シャドウ実体** (shadow object) と呼ぶ。シャドウ実体は、次節以降で述べる方法でのみ明示的な値の定義及び参照が可能である。

シャドウ領域でない記憶領域に割り付けられたデータで、シャドウ実体と同じ配列要素を表現するものを、そのシャドウ実体の**反映元**と呼ぶ。概念的に、一つのプロセッサにシャドウ実体とその反映元が同時にマップされることはない。シャドウ実体は、その反映元がマップされているプロセッサにはマップされない。また、概念的に、同一の配列要素に対応するシャドウ実体が、一つのプロセッサ上に複数マップされることはない。

【例】 ブロック分散のシャドウ実体

```
!HPF$ PROCESSORS P(4)
      REAL A(100)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
!HPF$ SHADOW A(1:2)
```

データ実体 A 及びそのシャドウ実体は、プロセッサ P に対して以下のようにマップされる。

プロセッサ	データ実体	シャドウ実体
P(1)	A(1), ···, A(25)	A(26), A(27)
P(2)	A(26), ···, A(50)	A(25), A(51), A(52)
P(3)	A(51), ···, A(75)	A(50), A(76), A(77)
P(4)	A(76), ···, A(100)	A(75)

【例】 フル SHADOW のシャドウ実体

```
!HPF$ PROCESSORS P(4)
      REAL A(100)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
!HPF$ SHADOW A(*)
```

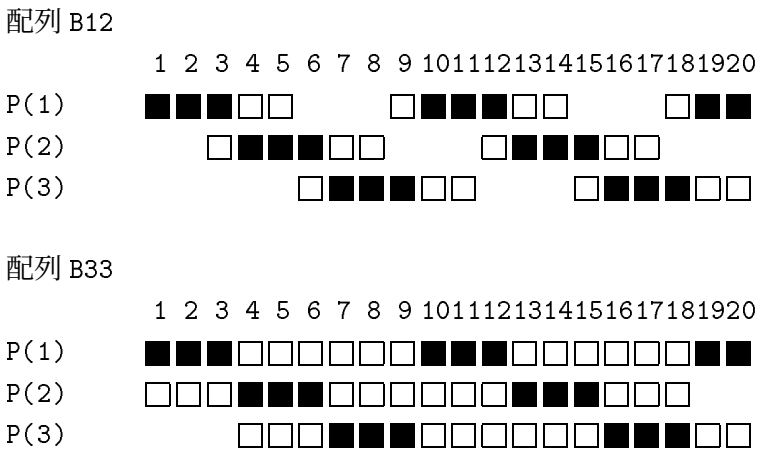
配列要素がデータ実体としてマップされているプロセッサ上には、おなじ配列要素はシャドウ実体としてマップされない。

プロセッサ	データ実体	シャドウ実体
P(1)	A(1), ···, A(25)	A(26), ···, A(100)
P(2)	A(26), ···, A(50)	A(1), ···, A(25), A(51), ···, A(100)
P(3)	A(51), ···, A(75)	A(1), ···, A(50), A(76), ···, A(100)
P(4)	A(76), ···, A(100)	A(1), ···, A(75)

【例】 サイクリック分散のシャドウ実体

```
!HPF$ PROCESSORS P(3)
      REAL,DIMENSION(20) :: B12,B33
!HPF$ DISTRIBUTE (CYCLIC(3)) ONTO P :: B12,B33
!HPF$ SHADOW B12(1:2),B33(3:3)
```

データ実体 A 及びそのシャドウ実体は、プロセッサ P に対して以下のようにマップされる。■はデータ実体としてのマッピング、□はシャドウ実体としてのマッピングを表す。



【仕様の根拠】 この例で分かるように、block-cyclic 分散では、シャドウ領域の大きさを大きくしていくと、1 つのプロセッサ上で同一配列要素に対するデータ実体とシャドウ実体を持ったり、シャドウ実体を複数持ったりする状態が考えられる (図 4.4 の a)。このような状態を認めると、LOCAL 指示 (4.6 節) によってアクセスするデータが一意に決まらないという問題が生じる。LOCAL 指示を拡張して、どのシャドウを選択するか指示できるようにすることも考えられるが、仕様が複雑になる。

この解決策として、利用者には以下のような見え方を提供するという方法が考えられる (図 4.4 の b)。

- ある配列要素に対してデータ実体を持つ場合、同じプロセッサ上では同じ配列要素に対して概念上はシャドウ実体を持たない。
- ある配列要素に対してシャドウ実体を持つ場合、そのシャドウ実体は概念上１つである。

このような見え方を言語仕様とすると、(a) のような実装を行う場合、重なり合うデータ実体とシャドウ実体の間の値の一致を処理系が保証しなければならない。しかしそのような実装では、LOCAL 指示やシャドウの明示の本来の目的である高速性を保証できなくなる。

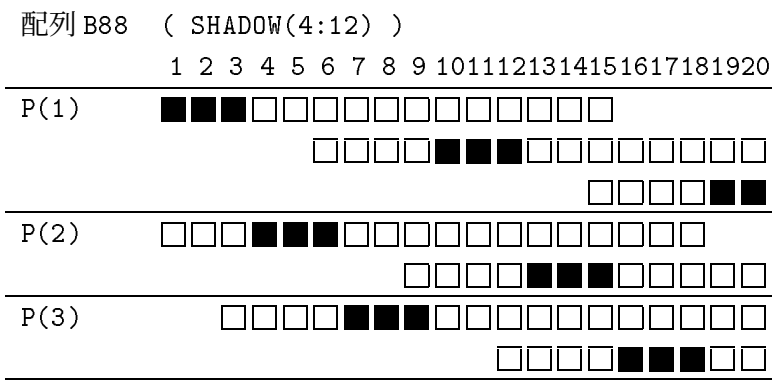
HPF/JA 仕様では、block-cyclic 分散のシャドウの大きさについて、このような重なりが起らない範囲に制約することにより (第 5 章)、言語仕様が複雑になることや、高速な実現ができない問題を回避している。

【以上】

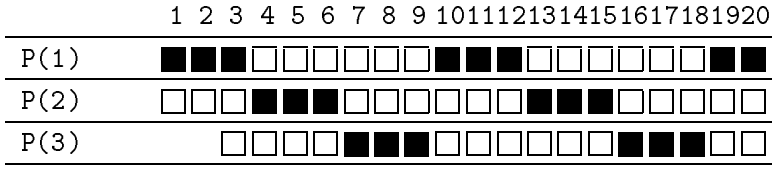
4.3.2　シャドウ実体の定義と参照

シャドウ実体は、以下のいずれかの方法を用いた場合に限り、値を定義したり参照したりすることができる。

1. 初期化
   シャドウ実体を初期化することはできない。シャドウ実体の初期値は常に不定である。
2. 値の定義



(a) シャドウ実体の重なりを許す割付け



(b) 重なりのあるシャドウ実体を認めない見せ方

図 4.4: シャドウ実体の重なり

(a) REFLECT 指示文 (4.4 節) による指定。
　反映元 (他のプロセッサが持つ) と同じ値が設定される。
(b) LOCAL 指示文による指定。
　LOCAL 指示文の有効範囲内での指定された変数への値の定義は、それぞれのプロセッサでその変数がデータ実体であるかシャドウ実体であるかに関わらず行われる。
ただし、LOCAL 指示の使用に関しては、4.6.3 項の制約が満たされていなければならない。

3. 値の参照
(a) LOCAL 指示文による指定。
　LOCAL 指示文の有効範囲内での指定された変数からの値の参照は、それぞれのプロセッサでその変数がデータ実体であるかシャドウ実体であるかに関わらず行われる。

【利用者への助言】 ON 指示文の EXT_HOME 節 (4.5 節) は、活動プロセッサを指定することだけが目的である。従って、EXT_HOME 節で指定された変数であっても、LOCAL 指示で指定しない限り、各プロセッサに閉じて定義または参照される保証はない。【以上】

4.3.3　シャドウ実体の確定と不定

シャドウ実体はデータ実体と同じように、確定または不定のどちらかの状態をもつ。不定であるシャドウ実体は、意味のある値を持つ保障はない。

データ実体を確定または不定にする動作は、標準 Fortran 規格の 14.7 節で規定されている。シャドウ実体を確定または不定にする動作は、以下で規定する。

1. シャドウ実体の初期状態
(a) シャドウ実体の初期状態は必ず不定である。
2. シャドウ実体が確定になる条件
　以下のいずれかの場合に限る。
(a) シャドウ実体は、4.3.2 項の 2 のいずれかにより定義すると確定になる。
(b) 手続の入口において、仮引数に付随するシャドウ実体は、下記のシャドウ継承条件を満たし、かつ対応する実引数のシャドウ実体が確定である場合、確定になる。
(c) 手続からの復帰直後において、実引数に付随するシャドウ実体は、下記のシャドウ継承条件を満たし、かつ対応する仮引数のシャドウ実体が、END 文又は RETURN 文の実行直前に確定である場合、確定になる。
3. シャドウ実体が不定になる条件
　以下のいずれかの場合に限る。
(a) 反映元と同じ活動プロセッサ集合内にあるシャドウ実体は、その反映元が不定になるとき不定になる。
(b) 反映元と同じ活動プロセッサ集合内にあるシャドウ実体は、LOCAL 指示なしでその反映元に値が定義されるとき、不定になる。
(c) 反映元と同じ活動プロセッサ集合内にあるシャドウ実体は、LOCAL 指示の有効範囲を出るとき、反映元と値が一致していなければ、不定になる。
(d) 反映元と異なる活動プロセッサ集合内にあるシャドウ実体は、反映元と同じ活動プロセッサ集合内に入るとき、反映元が不定であるか、反映元と値が一致していなければ、不定になる。

(e) シャドウ実体は、その親実体が再マッピングされるとき不定になる。

シャドウ継承条件

以下の条件が満たされるとき、シャドウ実体の値は実引数と仮引数の間で継承される。

1. 実引数のマッピングは仮引数のマッピングの特殊化 (HPF 仕様書 8.13 節) であること。
2. 仮引数は DYNAMIC 属性をもたないこと。

【実装者への助言】 この定義は、処理系に対して以下のことを保証している。

1. 反映元が更新されたり不定になった場合、対応するシャドウ領域に対して何もする必要はない。
2. 反映元とシャドウ実体が同一活動プロセッサ集合内にある場合、処理系は反映元の値をいつでもシャドウにコピーしてよい。
3. 再マッピング時にシャドウ領域の値は保証する必要はない。
4. 手続呼出し時／復帰時に再マッピングが不要な場合、シャドウ領域に対して何もする必要はない。再マッピングが必要な場合、シャドウ領域の値は保証する必要はない。

2. は、LOCAL 指示の使用に関する制約 (4.6.3 項) がユーザによって保証されていることを前提としている。【以上】

4.4 REFLECT 指示文

シャドウを持つ変数について、シャドウ実体に反映元の値を設定する。通信を非同期転送として行うこともできる。

4.4.1 書式

executable-directive-extended(H207) に *reflect-directive* を追加する。

J415 *reflect-directive* **is** [*async-prefix*] REFLECT *reflect-object-list*

J416 *reflect-object* **is** *object-name*

async-prefix は 4.1.4 項で定義する。

制約: *reflect-object* のデータ実体またはシャドウ実体を配置するプロセッサは、すべて活動プロセッサでなければならない。

【例】

```
REFLECT A
ASYNC(ID=ID1) REFLECT A,B,C
```

4.4.2　意味

reflect-object の持つすべてのシャドウ実体に、その反映元のデータ実体の値を複写する。*async-prefix* が指定されるとき非同期転送 (4.1 節) となる。

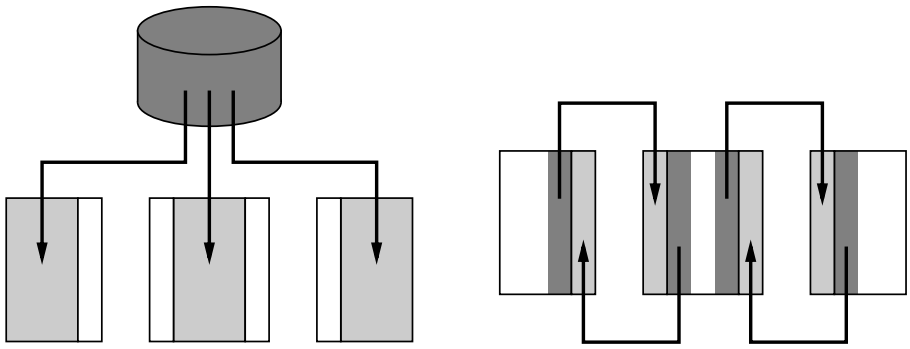
REFLECT 指示文の導入により、DO ループに INDEPENDENT 指示文を適用できる条件、つまり DO ループの繰返しが干渉しない条件 (HPF2.0 仕様書 5.1 節) には、以下の項目が追加される。

- ループ内で実行される REFLECT 指示文は、同じデータの参照、同じデータの再マッピング、および同じデータへの他の REFLECT 指示文と干渉する。

4.4.3　記述例

READ 文によりデータ実体へ値を読み込み、続く REFLECT 指示文によりシャドウ実体へ値をコピーする (図 4.5)。

```
!HPF$ PROCESSORS P(3)
      REAL A(M,N)
!HPF$ DISTRIBUTE (*,BLOCK) ONTO P :: A
!HPF$ SHADOW(0,1) :: A
      ...
      READ(*) A
!HPFJ REFLECT A
      ...
```



(a) READ(*) A　　(b) REFLECT A

図 4.5: REFLECT 指示文の例

4.5　ON 指示文の HOME 節の拡張

ON 指示文 (公認拡張機能) の HOME 節に変数を指定すると、変数の配置されているプロセッサだけを活動プロセッサにすることができるが、このときの配置とは変数のシャドウ領域を無

視したものである。ここでは、ON 指示文の HOME 節を拡張し、シャドウ実体の配置もまた変数の配置と考えた計算の分割を実現する。

4.5.1 書式

ON 指示の *home*(H907) を以下のように拡張する。なお、*on-stuff*(H903) は、4.6 節で拡張されている。

J417 *home-ja* **is** HOME (*variable*)
or HOME (*template-elmt*)
or EXT_HOME (*variable* [, *shadow-attr-stuff*])
or (*processors-elmt*)

制約: EXT_HOME 節の *shadow-attr-stuff* で指定される *shadow-spec-list* の長さは、*variable* の親実体の次元数と一致していなければならない。

制約: EXT_HOME 節の *shadow-attr-stuff* で指定される各次元の上下のシャドウ幅は、それぞれ *variable* の親実体の持つシャドウ幅と比較して等しいか小さくなければならない。

制約: *shadow-attr-stuff* の中の *shadow-spec-ja* は、*full-width*(アスタリスク) であってはならない。

【例】

```
EXT_HOME(A(I))
EXT_HOME(B(I+1),(2))
EXT_HOME(Z(I,:),(1:0,0))
```

4.5.2 意味

ON 指示文の *home* は、ON 指示文の有効範囲を実行する活動プロセッサ集合を指定する。*home* の 1 番目の書式 (HPF 公認拡張仕様) の意味は以下の通りである。

- *variable* がデータ実体として配置されているすべてのプロセッサで活動プロセッサを構成することを指定する。

これに対し、*home* の 3 番目の書式 (HPF/JA 拡張) の意味は以下の通りである。

- *shadow-attr-stuff* が省略されているとき、*variable* がデータ実体またはシャドウ実体として配置されているすべてのプロセッサで活動プロセッサを構成することを指定する。*variable* の親実体[4]がシャドウ領域を持たない場合、1 番目の書式と同じ意味となる。
- *shadow-attr-stuff* が指定されているとき、*variable* がデータ実体または *shadow-attr-stuff* で表現される範囲のシャドウ実体として配置されているすべてのプロセッサで活動プロセッサを構成することを指定する。

[4] 例えば *variable* が A(I) であれば全体配列 A のこと。

【仕様の根拠】 EXT_HOME 節の意味付けについて、以下の代案があった。

1. EXT_HOME 節を持つ ON 指示文の有効範囲内のすべての変数のアクセスには、明示的な LOCAL 指示がなければならないとする。(変数指定のない LOCAL 指示を使えば簡単に記述できる。) そうすれば、EXT_HOME 節を持つ ON 指示の有効範囲は常に高速に実行されることを保証することができる。
2. EXT_HOME 節を持つ ON 指示の有効範囲は、変数指定のない LOCAL 指示があるかのように扱う。つまり、EXT_HOME の意味に LOCAL の意味を含ませる。

しかし、以下の理由でこれらの意味付けは採用しなかった。

- EXT_HOME 節を持つ ON 指示の有効範囲内でも、プロセッサの外にあるデータを参照したいことがある。常に LOCAL 指示が必要では用途が限られる。
- HPF2.0 仕様の home 節の目的は活動プロセッサ集合を指定することだけであり、データの配置については何も言っていない。EXT_HOME に限ってデータ配置の意味を含ませるのは不自然である。

【以上】

EXT_HOME 節は HOME 節の簡便記法であると考えることができる。2 つの記述：

```
ON EXT_HOME(A(I),(M:N))
ON HOME(A(I-N:I+M))
```

は、以下の相違点を除けば同じ意味である。

- I の値として許される範囲が異なる。A の配列宣言の下限を l、上限を u とするとき、前者は $l \leq \mathtt{I} \leq u$ であるが、後者は $l + \mathtt{N} \leq \mathtt{I} \leq u - \mathtt{M}$ となる。
- 前者では、M，N の大きさはそれぞれ下と上のシャドウ幅を超えることはできない。
- *shadow-attr-stuff* を省略した EXT_HOME 節は、シャドウの大きさが分からない変数 (inherit 属性を持つ場合など) についても使用することができるが、HOME 節では同じことは表現できない。

【例】　次のプログラムの (*1) の部分で、I の値と活動プロセッサの対応は表のようになる。

```
!HPF$ PROCESSORS P(3)
      REAL A(9)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P
!HPF$ SHADOW(1) :: A
      ...
      DO I=1,9
!HPFJ  ON EXT_HOME(A(I))
         ...                ! (*1)
!HPF$  END ON
      END DO
```

I	活動プロセッサ集合
1	P(1)
2	P(1)
3	P(1), P(2)
4	P(1), P(2)
5	P(2)
6	P(2), P(3)
7	P(2), P(3)
8	P(3)
9	P(3)

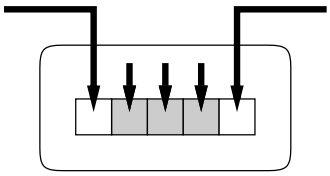
【利用者への助言】 この例に見られるように、EXT_HOME 節は活動プロセッサの変化を複雑にするため、大きく効率を落とすおそれがある。しかしながら、配列の初期化など、比較的簡単な計算の結果をシャドウ領域に直接設定したい場合には非常に有効な手段となる。EXT_HOME 節を使用して高い性能を得るためには、LOCAL 指示 (4.6 節) と併用することを強く推奨する (4.6.4.4 項参照)。【以上】

4.5.3 記述例

```
      REAL A(100)
!HPF$ DISTRIBUTE(BLOCK),SHADOW(1) :: A
```

(a) ON HOME + REFLECT
データ実体の値は計算により設定し、シャドウ実体の値は通信により設定する。

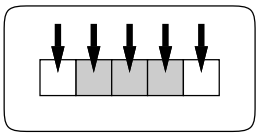
```
!HPF$ INDEPENDENT
      DO I=1,100
!HPF$   ON HOME(A(I))
        A(I)= ···
      END DO
!HPFJ REFLECT A
```



- 計算領域 (I の範囲) は隣接プロセッサと重なり合わない。
- シャドウ実体の値は隣接プロセッサからの通信によって得る。

(b) ON EXT_HOME
シャドウ領域を含む範囲について各プロセッサで計算する。

```
!HPF$ INDEPENDENT
      DO I=1,100
!HPFJ   ON EXT_HOME(A(I)),LOCAL(A(I))
        A(I)= ···
      END DO
```



- EXT_HOME 節の効果により、シャドウ実体として A(I) を持つプロセッサも計算に参加する。
- LOCAL 節の効果により、シャドウ実体への値の設定は隣接するプロセッサと協調して行われるのではなく、隣接するプロセッサと独立にそれぞれ行われる。

反映元からシャドウ実体への通信のコストと比較して、各要素の計算に必要なコストが小さいとき、シャドウ付きで活動プロセッサを指定する方法 (b) の方が効率がよい。

4.6 LOCAL 節と LOCAL 指示文

RESIDENT 節と RESIDENT 指示文 (公認拡張機能) は、データがその実行文を実行する活動プロセッサ内に存在することを宣言するものである。この情報によってコンパイラは、そのデー

タについて活動プロセッサの外部との通信を生成する必要がないことを知る。しかし、活動プロセッサが複数のプロセッサから成る場合、活動プロセッサ内部での通信は起こり得る。

ここで提案する LOCAL 節／指示文は、RESIDENT 節／指示文の考えをさらに進めて、指定したデータについて通信が一切必要ない、ということを指示するためのものである。

4.6.1　書式

4.6.1.1　ON 指示文の拡張

on-stuff(H903) を以下のように変更する。*home-ja* は 4.5 節で定義されている。

J418 *on-stuff-ja* **is** *home-ja* [, *on-optional-clause-list*]

J419 *on-optional-clause* **is** *resident-clause*
or *local-clause*
or *new-clause*

【仕様の根拠】 RESIDENT 節、NEW 節の出現順序を任意とし、新たに LOCAL 節を加える。
【以上】

【例】

```
ON HOME(B(I)), RESIDENT(A), LOCAL(A(I),A(I-1),A(I+1))
ON (PE(1:4)), NEW(I,J), LOCAL
```

4.6.1.2　LOCAL 節／指示文／指示構文

local-clause、*local-directive*、および *block-local-directive* を以下のように定義し、*executable-directive-extended*(H207) に追加する。

J420 *local-clause* **is** LOCAL *local-stuff*

J421 *local-stuff* **is** [(*local-object-list*)]

J422 *local-directive* **is** LOCAL *local-stuff*

J423 *local-construct* **is**
hpfja-directive-origin block-local-directive
block
hpfja-directive-origin end-local-directive

J424 *block-local-directive* **is** LOCAL *local-stuff* BEGIN

J425 *end-local-directive* **is** END LOCAL

J426 *local-object* **is** *object*

構文規則は、キーワードが LOCAL であることを除けば RESIDENT のものと同じである。*local-object* は、字面上の一致によって指定する。

【例】 LOCAL 節

```
LOCAL(A(I),S)
LOCAL
```

【例】 LOCAL 指示文による指定

```
!HPFJ LOCAL(A(I),S)
      A(I) = S
```

【例】 LOCAL 指示構文による指定

```
!HPFJ LOCAL BEGIN
      A(I) = S
      CALL SUB(A,I,S)
!HPFJ END LOCAL
```

4.6.2 意味

ON 指示文の LOCAL 節、および LOCAL 指示文は、その指示文の有効範囲内において、指定した変数の値や属性の参照と、値の定義を、以下のように行うことを指示する。

参照 すべての活動プロセッサは、それぞれのプロセッサが持つ変数のコピーから読み出す。他のプロセッサが持つ変数のコピーは参照しない。

定義 すべての活動プロセッサが持つ変数のコピーに、それぞれのプロセッサで書き込む。

各プロセッサで参照または定義されるコピーは、データ実体であるかもしれないし、シャドウ実体であるかもしれない。

変数の指定のない LOCAL 指示は、その有効範囲内のすべての変数の参照と定義 (有効範囲内で直接または間接に呼び出される手続内のすべての変数を含む) に対して LOCAL と指示した場合と同じ効果を持つ。

RESIDENT 指示と同様に、LOCAL 指示は、それを囲む ON 指示と、対象となる変数のマッピングに関係した表明である。処理系が ON 指示とマッピングの指示に従わない場合、LOCAL 指示は正しい意味をもたない。

【仕様の根拠】 LOCAL 指示の目的は、指定したアクセスについてプロセッサ間通信なして正しく行えることを、ユーザが保証することである。【以上】

4.6.3　制約

1. LOCAL 指示で指定された変数のコピーは、活動プロセッサ内のすべてのプロセッサに配置されていなければならない。(それはデータ実体であってもよく、シャドウ実体であってもよい。)
2. LOCAL 指示で指定された変数に定義がある場合、指定された変数のコピーは、活動プロセッサ集合に含まれないプロセッサには、データ実体として配置されていてはならない。(活動プロセッサ外にシャドウ実体として配置されているのはよい。)
3. LOCAL 指示で指定された変数について、指定の有効範囲内で以下のことを行ってはならない。
   - 再マッピング
   - 領域の割付けまたは解放
   - REFLECT 指示文の *reflect-object* になること
   - REDUCTION 節による指定
   - RESIDENT 節または指示文による指定
4. 変数指定のない LOCAL 指示の有効範囲内で呼び出される手続は、グローバル手続 (グローバルモデル) であってはならない。
5. LOCAL 指示の有効範囲から出たとき、及び、有効範囲の中で指定された ON 指示の有効範囲から出たとき、指定した変数のデータ実体のコピーは活動プロセッサ内で値が一致していなければならない。(シャドウ実体の値はこれと一致している必要はない。ただし、一致していない場合にはシャドウ実体の値はそのとき不定となる (4.3.3 項参照)。)

【仕様の根拠】 制約 1,2 の目的

1. により、変数の参照ではそれを行うプロセッサがそのデータを所有していることが保証されるので、処理系は通信が必要である可能性の検査を省略してプロセッサ上のデータを参照することができる。また同様に、変数の定義ではそれを行うプロセッサが必ずそのデータ領域を確保していることが保証されるので、処理系は不正な領域を書きつぶす可能性の検査を省略して値を設定することができる。

2. により、活動プロセッサ内での変数の定義によりすべてのデータ実体のコピーに漏れなく値が設定されることが保証されるので、処理系は変数の値の一致を保証するための通信を生成する必要がなくなる。【以上】

4.6.4　記述例

4.6.4.1　LOCAL 指定された変数の参照

```
      SUBROUTINE SUB1(A,B)
!HPF$ INHERIT A,B                        ! A,B ともコンパイル時には分散不明
      …
      DO I=1,100
!HPFJ   ON HOME(A(I)), LOCAL(B(I))       ! B(I) の参照に通信は不要であると宣言
        A(I) = B(I)
```

```
      END DO
```

ON 指示文で指定される活動プロセッサと変数 B(I) の配置が、図 4.6 の (a)、(b)、または (c) に示す関係にある場合に、このプログラムは正しい。このような記述がある場合、変数 B(I) は通信なしで参照されることが保証される。(d) に示す関係にある場合は、このプログラムの正常な動作は保証されない。

なお、(d) のような場合には、LOCAL(B(I)) を RESIDENT(B(I)) に置き換えると正しいプログラムとなる。

4.6.4.2 LOCAL 指定された変数の定義

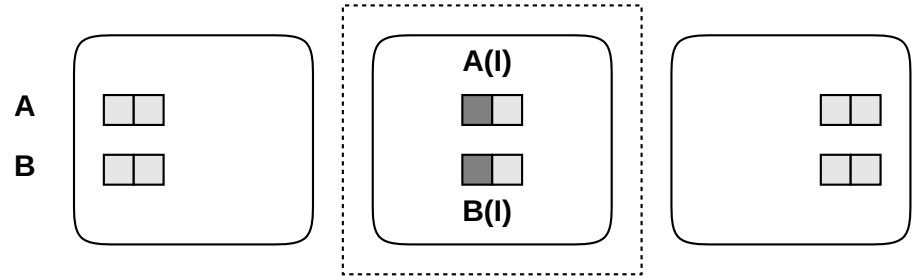
```
      SUBROUTINE SUB1(A,B)
!HPF$ INHERIT A,B                    ! A,B ともコンパイル時には分散不明
      …
      DO I=1,100
!HPFJ   ON HOME(A(I)), LOCAL(B(I))   ! B(I) の定義に通信は不要であると宣
言
        B(I) = A(I)
      END DO
```

ON 指示文で指定される活動プロセッサと変数 B(I) の配置が、図 4.6 の (a) または (b) に示す関係にある場合に、このプログラムは正しい。このような記述がある場合、変数 B(I) は通信なしで定義されることが保証される。(c) または (d) に示す関係にある場合は、このプログラムの正常な動作は保証されない。

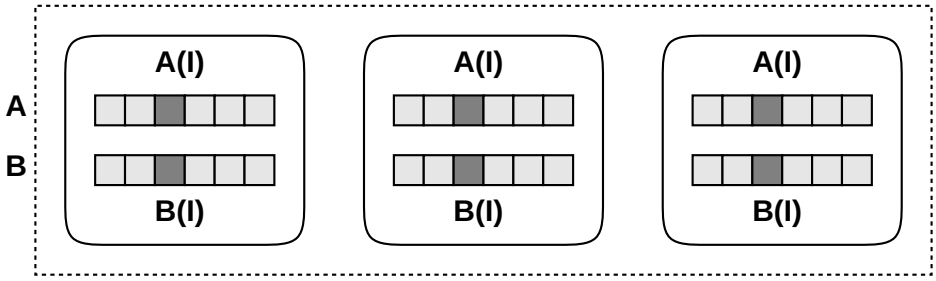
なお、(d) のような場合には、LOCAL(B(I)) を RESIDENT(B(I)) に置き換えると正しいプログラムとなる。しかし、(c) のような場合には、RESIDENT(B(I)) に置き換えても正しいプログラムとはならない。

4.6.4.3 Shadow を持つ変数の LOCAL 指定

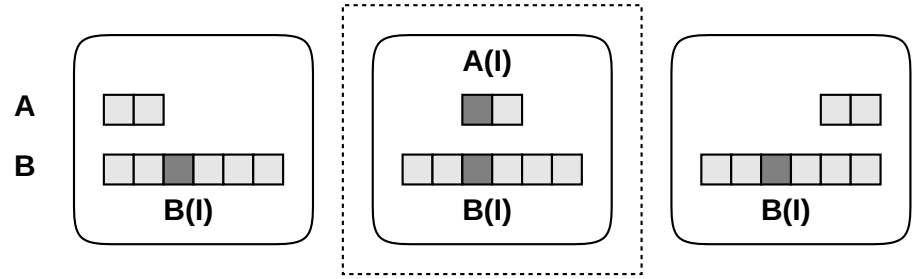
```
!HPF$ PROCESSORS P(4)
      REAL A1(400),A2(400),B(400)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A1,A2,B
!HPF$ SHADOW(1) :: A1,A2,B
      …
!HPFJ REFLECT B          ! B のシャドウを参照可能にする。
!HPF$ INDEPENDENT
      DO I=2,399
!HPFJ   ON HOME(A1(I)), LOCAL(B,A2(I)) BEGIN
          A1(I)=B(I-1)+B(I)+B(I+1)
          A2(I)=B(I-1)+2*B(I)+B(I+1)
!HPFJ   END ON
```



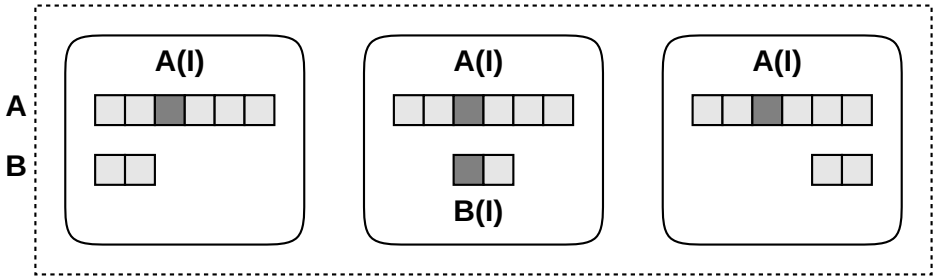
(a) B(I) の配置が活動プロセッサ (単一) と一致する場合



(b) B(I) の配置が活動プロセッサ (複数) と一致する場合



(c) すべての活動プロセッサに B(I) が配置され、
活動プロセッサ外にも B(I) がある場合



(d) 活動プロセッサの一部にだけ B(I) が配置される場合

図 4.6: 活動プロセッサと変数配置の関係
点線内は `ON HOME(A(I))` 指定時の活動プロセッサ

```
      END DO
```

変数 B は LOCAL 指定されているため、B(I-1)、B(I)、及び B(I+1) の参照は各プロセッサ内で行われる。I の値がブロック分散区間の下限値であるとき、B(I-1) の参照は下側シャドウ領域の参照となる。同様に、I の値がブロック分散区間の上限値であるとき、B(I+1) の参照は上側シャドウの参照となる。

A2(I) もまた LOCAL 指定されているため、各プロセッサ内で値の定義が行われる。すべての I について、A2(I) をシャドウ実体として配置するプロセッサは活動プロセッサにならないので、A2 のシャドウ領域には値が設定されない。

4.6.4.4 シャドウ付き ON 指定時の LOCAL 指定

```
!HPF$ PROCESSORS P(4)
      REAL A1(400),A2(400),B(400)
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A1,A2,B
!HPF$ SHADOW(1) :: A1,A2,B
      ...
!HPF$ INDEPENDENT
      DO I=2,399
!HPFJ   ON EXT_HOME(A1(I)), LOCAL BEGIN
          A1(I)=B(I)
          A2(I)=2*B(I)
!HPFJ   END ON
      END DO
```

EXT_HOME 節を持つ ON 指示により、各プロセッサが活動プロセッサとして実行する I の範囲は、A1(I) をシャドウ領域として配置する範囲にまで拡大されている。従って、例えば、プロセッサ P(2) は区間 $100 \leq I \leq 201$ について活動プロセッサとなるので、$I = 100$ 及び $I = 201$ のときには B(I) の参照はシャドウ実体の参照となり、A1(I), A2(I) の定義はシャドウ実体への値の設定となる。

4.6.5 RESIDENT と LOCAL の比較［参考］

4.6.5.1 シャドウを持たない変数に関して

RESIDENT は、変数の参照と定義で必要となる通信が活動プロセッサ集合内に閉じることを表現するのに対し、LOCAL は、変数の参照と定義が各プロセッサ内に閉じる（通信が一切必要ない）ことを表現する。どちらも、活動プロセッサ集合の内と外で同一変数の値が一致しなくなる状態を避けるため、定義される変数についてはコピーはすべて活動プロセッサ集合内になければならないとしている。

以上の性質は、表 4.1 のようにまとめることができる。同表に示した包含関係から以下のことが言える。

1. 活動プロセッサが 1 プロセッサから成るとき、RESIDENT と LOCAL の意味は同じである。
2. LOCAL は RESIDENT の意味を包含している。つまり、LOCAL(v) が正しい指示であるなら、RESIDENT(v) に置き換えても常に正しい。

4.6.5.2　Shadow を持つ変数に関して

RESIDENT は、シャドウ領域の配置を考慮に入れない。つまり、活動プロセッサに変数がシャドウ実体として存在していても RESIDENT と指定することはできない。一方 LOCAL は、シャドウ領域の配置も対象となる。LOCAL では活動プロセッサ集合内のすべてのプロセッサに変数が配置されていなければならないが、シャドウ実体としての配置であってもよい。

RESIDENT においても LOCAL においても、データ実体の値がプロセッサ間で一致しなくなることを避けるため、値を定義する場合には活動プロセッサ集合の外にデータ実体が配置されていてはならない。しかしシャドウ実体については、値の正しさはユーザが保証するという立場から、活動プロセッサ集合の外に存在してもよいとし、プロセッサ間で一致しない状態（データ実体は更新されるが対応するシャドウ実体には反映されない状態）が起こり得る。

以上の性質は、表 4.2 のようにまとめることができる。

4.7　通信スケジュール再利用

本指示文の目的は、並列ループ中に現れた配列データに対する非定型なパターンを有するデータアクセスに対する通信を効率化することである。有限要素法や、スパース行列処理は、実用の数値シミュレーションコードで頻繁に用いられるが、HPF2.0 においては、公認拡張を含めても、コンパイラが効率のよい通信を生成できるコードを記述することが困難である。本指示文は、通信パターンの再利用可能性をプログラムに記述する手段を与えることにより、コンパイラによる効率のよい通信生成を可能とすることを目的とする。基本的な考え方は、スイス CSCS で R. Ruehl らによって考案され、Vienna 大学の Vienna Fortran でも採用されている。

非定型配列参照で用いられる配列の間接参照では、通信パターンが実行時にしか定まらない。従来のコンパイラは、これらの間接参照について、対象とする配列全範囲のデータをすべてのプロセッサにばらまくような通信を生成するか、または、一要素ごとの通信を生成するものが大半であり、これが、非定型処理の並列実行性能を劣化させる大きな要因となっている。

非定型通信をループの前後でパッキングして、効率よく行う手法としては、Maryland 大学の J. Saltz らによって提唱された Inspector-Executor 法がある。ループ処理を、非定型配列アクセスのインデックスを求める処理 (Inspector と呼ばれる) と求めたインデックスを元にして通信を行う部分、ループ処理 (Executor と呼ばれる) に分け、通信のパッキングを行うものである。しかしながら、Inspector の処理オーバヘッドが大きく、通常の場合には、並列化による速度向上が得られないという問題がある。

一方で、有限要素法などのシミュレーションでは、配列間接参照において、同一のデータ参照パターンを何度も繰り返し用いることが多い。外側の時間発展ループや、収束ループを何度も実行する際に、ある一定期間データ構造が変化しないためである。本指示文では、同

表 4.1: シャドウを持たない変数の RESIDENT 指定と LOCAL 指定

	RESIDENT(v)	LOCAL(v)
v が参照できる条件	活動プロセッサ集合内の少なくとも１つのプロセッサに v がある。 $P \cap H(v) \neq \phi$	活動プロセッサ集合内のすべてのプロセッサに v がある。 $P \subseteq H(v)$
v が定義できる条件	活動プロセッサ集合内の少なくとも１つのプロセッサに v があり、かつ、活動プロセッサ集合の外には v はない。 $P \supseteq H(v)$	活動プロセッサ集合内のすべてのプロセッサに v があり、かつ、活動プロセッサ集合の外には v はない。 $P = H(v)$

v は変数 (名前付きデータ実体、配列要素、部分配列、構造体成分、または文字部分列)。
P は活動プロセッサ集合。
$H(v)$ はデータ実体として v が配置されているプロセッサの集合。

表 4.2: シャドウを持つ変数の RESIDENT 指定と LOCAL 指定

	RESIDENT(v)	LOCAL(v)
v が参照できる条件	活動プロセッサ集合内の少なくとも１つのプロセッサにデータ実体としての v がある。 $P \cap H(v) \neq \phi$	活動プロセッサ集合内のすべてのプロセッサにデータ実体またはシャドウ実体としての v がある。 $P \subseteq H^+(v)$
v が定義できる条件	活動プロセッサ集合内の少なくとも１つのプロセッサにデータ実体としての v があり、かつ、活動プロセッサ集合の外にはデータ実体としての v はない。 $P \supseteq H(v)$	活動プロセッサ集合内のすべてのプロセッサにデータ実体またはシャドウ実体としての v があり、かつ、活動プロセッサ集合の外にはデータ実体としての v はない。 $H(v) \subseteq P \subseteq H^+(v)$

v は変数 (名前付きデータ実体、配列要素、部分配列、構造体成分、または文字部分列)。
P は活動プロセッサ集合。
$H(v)$ はデータ実体として v が配置されているプロセッサの集合。
$H^+(v)$ はデータ実体またはシャドウ実体として v が配置されているプロセッサの集合。($H^+(v) \supseteq H(v)$)

一の通信パターンが何度も現れることをコンパイラに伝える手段を与えることにより、これら非定型配列アクセスの通信生成を効率化することを目的としている。

4.7.1　書式

executable-directive-extended(H207) に *index-reuse-directive* を追加する。

J427　*index-reuse-directive*　　**is**　INDEX_REUSE [(*scalar-logical-expr*)]
index-reuse-variable-list

J428　*index-reuse-variable*　　**is**　*array-variable-name*

【例】

```
      LOGICAL REUSE
      ...
      REUSE = .FALSE.
!HPFJ INDEX_REUSE (REUSE) A, B
```

4.7.2　意味

INDEX_REUSE 指示文の直後の DO ループまたは FORALL ループを、その INDEX_REUSE 指示文の対象ループと呼ぶ。*scalar-logical-expr* の値が真のとき、対象ループに対して以下の条件がすべて成り立っていることをプログラマが処理系に対して保証する。

- 対象ループの範囲中の各ループの反復回数が、（逐次実行順序で）前回実行した時の反復回数と同一である。
- *index-reuse-variable-list* に指定した配列の対象ループ中でのすべての出現に対して、対象ループの範囲中のすべてのループの反復における、部分名、各次元の添字の値、及び部分列範囲の値が、対象ループを（逐次実行順序で）前回実行した時の対応する反復での値と同一である。
- 対象ループが条件分岐などの制御構造を含み、*index-reuse-variable-list* に指定した配列のいずれかの出現がその制御を受ける場合、対象ループのすべての反復における制御の流れが、対象ループを（逐次実行順序で）前回実行した時の制御の流れと同一である。

このとき、処理系はその他の条件を勘案し、可能であれば前回（またはそれ以前）の対象ループの実行時に構成された当該配列に関する通信スケジュールを再利用する。

scalar-logical-expr の値が偽のとき、上記の条件のいずれかが成り立たないかもしれないことをプログラマが処理系に対して言明する。このとき、処理系は当該配列に関する通信スケジュールを再構成し、保存する。

scalar-logical-expr が省略されたとき、*scalar-logical-expr* に.TRUE. が指定されたのと同値とする。

ループを初めて実行するときは、*scalar-logical-expr* の値は無視され、処理系はつねに当該配列に関する通信スケジュールを構成し、保存する。

保存された通信スケジュールは SAVE 属性を持つ変数と同様に振舞う。すなわち、INDEX_REUSE 指示文を含む手続の実行を一旦終わり、再びその手続が呼び出されたときにも、対応する通信スケジュールは再利用できる。

【仕様の根拠】 一般に、ループ内で不規則アクセスされる配列の通信スケジュールが再利用できるためには、そのループに関して以下の条件が成り立たなければならない。

1. 当該配列のマッピングが前回の実行時と同一である。
2. ループの計算マッピング（ループの各反復の実行のプロセッサへの割当て）が前回の実行時と同一である。
3. 当該配列の上下限が前回の実行時と同一である。
4. ループの上下限、増分値が前回と同一である。
5. ループの各反復で、当該配列の参照に係る制御の流れが前回と同一である。
6. ループの各反復での当該配列の添字の値が前回と同一である。

しかし、処理系の実装方法によっては、このほかに以下のような条件が必要な場合が考えられる。

7. 当該配列が共通ブロック中の変数またはモジュール変数である場合、対象ループ中から呼び出される手続中では参照または定義されてはならない。
8. 当該配列は対象ループ中から呼び出される手続の実引数であってはならない。
9. ループ中から対象ループを含む手続を再帰呼び出ししてはならない。
10. ループはタスクリージョン内にあってはならない。
11. ループ中に *index-reuse-variable-list* 中の変数に起因する依存関係があってはならない。
12. その他。

これらの条件のうちいずれが実際に必要かは処理系の実装方法によって大きく異なり、言語仕様として統一することは困難である。

そこで、これらの条件のうち、処理系による自動判定が困難なもののみをプログラマに指示させ、その他の必要条件は処理系が自身で判断するのが妥当であると考えられる。

具体的には、本指示文により上記 5, 6 の成立のいかんをプログラマから処理系に指示させることとした。また、上記 7 は制約とした。【以上】

【実装者への助言】 上記 5, 6, 7 以外の条件は処理系が自身で判断し、可能な限り通信スケジュールの再利用を行うことを強く推奨する。【以上】

4.7.3 制約

- INDEX_REUSE 指示文の直後の実行文は、DO 文、FORALL 文および構文でなければならない。
- *index-reuse-variable-list* 中の配列が共通ブロック中の変数またはモジュール変数である場合、または親子結合により手続外でアクセス可能な変数である場合、INDEX_REUSE 指示文に続く DO ループまたは FORALL ループから呼び出される手続中ではその変数は参照も定義もされてはならない。

4.7.4　記述例

【例】

```
!HPF$ DISTRIBUTE A(BLOCK)
      LFLAG=.FALSE.
      DO ITIME=1,IT ！時間発展ループなど
        ...
!HPFJ   INDEX_REUSE (LFLAG) A
!HPF$   INDEPENDENT, NEW(IDX,IDX2)
        DO I=1,N
          IDX = IBANK(I)
          IDX2 = IBANK2(I)
          B(I) = A(IDX) + C(I)
          IF (D(I).LT.C(I)) THEN ! 論理式の真偽は前回の対応する実行と同一
            D(I) = A(IDX2)
          END IF
        END DO
        ...
        IF（データ構造変更必要）THEN
          （データ構造変更コード）
          LFLAG=.FALSE.          ! 次の反復では通信スケジュール再利用しない
        ELSE IF
          LFLAG=.TRUE.           ! 再利用する
        END IF
        ...
      END DO
```

INDEX_REUSE 指示文により、LFLAG が真であるとき、「DO I」のループ中にある IF 文の条件節の真偽が対応する反復ごとに同一であること、および A の添字 IDX と IDX2 の値が対応する反復ごとに不変であることを処理系に対して保証している。

このケースでは ON 指示文が明示的に指定されていないため、処理系が計算マッピングをループの実行ごとに変化させず、また A の分散が変化しないことを保証できる場合に、A の通信スケジュールが再利用できる。

第5章　HPF2.0に対する制限および変更

HPF/JAではHPF2.0に対していくつかの制限と変更を行う。この主な目的は以下である。

- 優先順位の低い機能を制限とすることによって処理系の早期実装を促進する。
- HPF2.0の仕様上曖昧な部分を、制限としたり明確に定義したりすることによって、プログラムの実行結果が処理系依存となるのを防ぐ。

これらの制限、特に上記の第1の目的による制限はHPF/JAの将来のバージョンでは解除される予定である。

5.1　HPF2.0に対する制限

HPF/JAでは、HPF2.0に対して、以下の制限を設ける。

- ポインタおよび指示先のマッピング
  HPF/JAでは、ポインタおよび指示先のマッピングはできない。HPF2.0仕様書で言うと、8.8節全体の記述、および、仕様書の他の部分におけるポインタおよび指示先のマッピングに関する記述のすべてを無効とする。
- 構造体成分のマッピング
  HPF/JAでは、構造体成分のマッピングはできない。HPF2.0仕様書で言うと、8.9節全体の記述、および、仕様書の他の部分における構造体成分のマッピングに関する記述のすべてを無効とする。

  【仕様の根拠】 ポインタ、指示先、構造体成分のマッピングを実装するためには、他の変数をマッピングする場合と比べて余分な変換が必要となる。例えば、HPF2.0仕様書の8.8節や8.9節に述べられているような様々な仕様を実装しなければならない。

  一方、ユーザの立場で見ると、ポインタ、構造体などのFortran 90から導入された機能は、日本ではまだ広く利用されているわけではない。したがって、これらに対するマッピングに制限を設けたとしても、多くのユーザはあまり不便を感じない。むしろ、よく使う機能を早期に実現してもらう方が望ましい。

  以上の理由により、HPF/JAの本バージョンでは、ポインタ、指示先、構造体成分に対するマッピングを制限とすることにした。【以上】

- INDIRECT分散形式
  HPF/JAでは、分散形式にINDIRECTを指定することはできない。HPF2.0仕様書で言うと、8.10節の構文H810の右辺から、INDIRECTの行を削除する。また、仕様書の他の部分におけるINDIRECT分散形式に関する記述のすべてを無効とする。

【仕様の根拠】INDIRECT 分散を実装するには、一般には実行時に巨大なインデックス変換テーブルが必要となる。さらにこのテーブルに対するアクセスを高速に行おうとすれば、各プロセッサにテーブル全体のコピーを置くことになり、スケーラビリティを損なう。

この問題を改善し INDIRECT 分散で性能を出すためには、inspector/executor 法が必須になるが、この方法も現在は十分成熟しているとは言えない。結局、INDIRECT 分散を用いた場合、性能の見通しが立たない。

以上の理由により、HPF/JA の本バージョンでは INDIRECT 分散は制限とすることにした。【以上】

- RANGE 指示文
  HPF/JA では、RANGE 指示文は利用できない。HPF2.0 仕様書で言うと、8.11 節全体の記述、および、仕様書の他の部分における RANGE 指示文に関する記述のすべてを無効とする。

  【仕様の根拠】 RANGE 指示文が最も有用なのは、分散形式が INDIRECT となりえないことを処理系に知らせる場合である。INDIRECT 分散の可能性があると、処理系は非常に効率の悪いコードを生成せざるを得ない。

  しかし、INDIRECT 分散形式は、HPF/JA の本バージョンでは制限となったので、RANGE 指示文の必要性は少なくなった。そこで、RANGE 指示文も制限とすることにした。

  将来、INDIRECT 分散を採用する場合は RANGE 指示文も必要になる可能性があり、これらを一緒に議論するべきである。【以上】

- HPF_GLOBAL, Fortran_LOCAL 以外の外来手続
  HPF/JA では、HPF_GLOBAL, Fortran_LOCAL 以外の外来手続は利用できない。HPF2.0 仕様書で言うと、第 11 章の記述の中で HPF_GLOBAL、Fortran_LOCAL に関するものだけを有効とする。

  【仕様の根拠】 処理系の早期実現のため、ローカルモデルとして最低限必要と考えられる Fortran_LOCAL、および HPF 自身である HPF_GLOBAL のみを採用することにした。【以上】

- CYCLIC 分散に対するシャドウ幅の制限
  HPF/JA では、配列の CYCLIC 分散されている次元に対するシャドウ幅は、次の条件を満たしていなければならない。

$$(w_l + w_h) \times |s| \leq m \times (p-1)$$

ここで、

w_l	下端シャドウ幅
w_h	上端シャドウ幅
s	最終的な整列先に対する整列の刻み幅
m	最終的な整列先の分散におけるブロックの大きさ
p	プロセッサ構成の寸法

ただし、s、m、p は対象としている配列次元に対応するものである。
フル SHADOW はこの制限の対象外である。

【仕様の根拠】 この制限は、同一配列要素が 1 プロセッサのシャドウ領域内に複数存在しないための条件である。4.3 項を参照。【以上】

- 非同期入出力
  HPF/JA では、HPF 公認拡張で規定されている非同期入出力は利用できない。HPF2.0 仕様書で言うと、第 10 章全体の記述、および、仕様書の他の部分における非同期入出力に関する記述のすべてを無効とする。

  【仕様の根拠】 非同期入出力は、HPF が対象とする分散メモリ型の並列計算機に特有の話題ではないので、処理系の早期実現のために制限とした。【以上】

5.2 HPF2.0 に対する変更

HPF/JA では、HPF2.0 の一部を以下のように変更する。

- DYNAMIC 変数の手続戻り時のマッピング
  HPF/JA では、手続の実引数と仮引数が共に DYNAMIC 属性であったとしても、手続内で実行された仮引数のリマッピングは、実引数のマッピングには反映されないものとする。
  これはすなわち、HPF2.0 仕様書で言うと、8.6 節の第 1 項目の第 2 段落全体「手続が呼出し元に戻った後の仮引数の再分散に関する影響は ･･･ RANGE 指示文で指定された分散形式の一つにマッチしなければならない。」を、以下のように変更することである。
  「手続が呼出し元に戻った後は仮引数の再分散の影響は残らない。実引数のマッピングは呼出し前のマッピングのままである。これは、引数の再マッピングは、呼出し元からは見えないという HPF2.0 仕様書 4.2 節の原理に従っている。」

  【仕様の根拠】 HPF2.0 では、実引数と仮引数が共に DYNAMIC 属性であった場合には、仮引数のリマッピングを実引数に反映させることになっている。しかしある配列が呼び先でリマップされた場合、呼び元でその配列と整列関係にあった他の配列がどうなるかという点があいまいである。例えば次のようなプログラムで問題が生じる。

```
      PROGRAM EX1
      REAL A(10),B(10)
!HPF$ DYNAMIC A,B
!HPF$ ALIGN A(I) WITH B(I)
!HPF$ DISTRIBUTE B(BLOCK)
         :
      CALL SUB(B)
         :
      END
```

```
      SUBROUTINE SUB(B)
      REAL B(10)
!HPF$ DYNAMIC B
!HPF$ DISTRIBUTE B(BLOCK)
        :
!HPF$ REDISTRIBUTE B(CYCLIC)
        :
      RETURN
      END
```

この例では、手続SUB内で配列BがBLOCKからCYCLICにリマップされている。戻り時にこれが呼び元に反映されるとすると、配列Bに整列していた配列Aは、やはりCYCLICにリマップされるべきなのかどうかという問題が生じる。

さらに問題になるのは次の例のように配列Aの方がリマップされる場合である。

```
      PROGRAM EX2
      REAL A(10),B(10)
!HPF$ DYNAMIC A,B
!HPF$ ALIGN A(I) WITH B(I)
!HPF$ DISTRIBUTE B(BLOCK)
        :
      CALL SUB(A)
        :
      END

      SUBROUTINE SUB(A)
      REAL A(10)
!HPF$ DYNAMIC A
!HPF$ DISTRIBUTE A(BLOCK)
        :
!HPF$ REDISTRIBUTE A(CYCLIC)
        :
      RETURN
      END
```

呼び先でのCYCLIC分散が呼び元にも引き継がれるとすると、配列AとBの整列関係が壊れてしまうという問題が生じる。

さらに、呼び先で局所変数やテンプレートに再整列された場合、呼び元に戻った後は整列先がなくなってしまうという問題もある。

以上のような問題に対してHPF2.0仕様書には明確な記述がなく、このままでは処理系依存の解釈となりかねない。これを避けるため、HPF/JAでは暫定仕様として、DYNAMIC属性の有無にかかわらず、手続からの戻り時には呼び元でのマッピングに戻すことにした。【以上】

- 仮引数に対する NEW, REDUCTION 指示
  HPF/JA では、仮引数に対する NEW, REDUCTION 指示を許す。HPF2.0 仕様書で言うと、5.1 節の第 4 の制約から第 1 項目「仮引数」を削除する。

  【仕様の根拠】上記の第 4 の制約の目的は、NEW 変数や REDUCTION 変数が他の変数と別名関係にないことを保証するためと考えられる。しかし仮引数に関しては、Fortran の仕様により、別名問題を起こさないようにいくつかの制約が設けられている。例えばもし他の変数と別名関係にある場合は、その仮引数を用いずに値を代入してはならないことになっている。したがって、仮引数を NEW 変数や REDUCTION 変数としても別名問題は起こらないので安全である。

  また、実プログラムにおいては、仮引数を NEW 変数としたい状況がしばしば起こる。それは、一時的な作業領域として使われる配列を呼び元で確保しておき、引数として手続に渡す場合である。このような一時的配列は NEW 変数として使いたい場合が多い。

  以上の理由により、HPF/JA では、仮引数に対する NEW、REDUCTION 指示を許すことにした。【以上】

附属書 A　構文規則

この附属書は、本 HPF/JA 言語仕様書で定義される構文規則の一覧である。

A.2　記述法と構文

A.2.2　指示文の構文

J201　*hpfja-directive-line*　**is**　*hpfja-directive-origin*　*hpf-directive*

J202　*hpfja-directive-origin*　**is**　`!HPFJ`
or　`CHPFJ`
or　`*HPFJ`

J203　*specification-directive-ja*　**is**　*processors-directive*
or　*subset-directive*
or　*align-directive*
or　*distribute-directive*
or　*inherit-directive*
or　*template-directive*
or　*combined-directive*
or　*sequence-directive*
or　*dynamic-directive*
or　*shadow-directive*
or　*asyncid-directive*

J204　*executable-directive-ja*　**is**　*independent-directive-ja*
or　*realign-directive-ja*
or　*redistribute-directive-ja*
or　*on-directive*
or　*resident-directive*
or　*asynchronous-directive*
or　*asyncwait-directive*
or　*reflect-directive*
or　*local-directive*
or　*index-reuse-directive*

J205 *executable-construct-ja* **is** *action-stmt*
or *case-construct*
or *do-construct*
or *if-construct*
or *where-construct*
or *on-construct*
or *resident-construct*
or *task-region-construct*
or *asynchronous-construct*
or *local-construct*

A.3 並列処理指定に関する HPF/JA 拡張

A.3.1 REDUCTION 種別の指定

J301 *independent-directive-ja* **is** `INDEPENDENT` [`,` *new-clause*]
[`,` *reduction-clause-ja-list*]

J302 *reduction-clause-ja* **is** `REDUCTION`
`(` [*reduction-kind* `:`] *reduction-spec-list* `)`

J303 *reduction-kind* **is** *reduction-operator*
or *reduction-function*
or *maxmin-kind*

J304 *reduction-operator* **is** `+`
or `*`
or `.AND.`
or `.OR.`
or `.EQV.`
or `.NEQV.`

J305 *maxmin-kind* **is** `FIRSTMAX`
or `FIRSTMIN`
or `LASTMAX`
or `LASTMIN`

J306 *reduction-spec* **is** *reduction-variable* [`/` *location-variable-list* `/`]

J307 *location-variable* **is** *scalar-variable-name*

制約: *reduction-kind* が *maxmin-kind* であるとき、*reduction-spec* は *location-variable-list* を持たなければならない。*reduction-kind* が *maxmin-kind* でないとき、または *reduction-kind* が省略されたときは、*reduction-spec* は *location-variable-list* を持ってはならない。

制約: *reduction-kind* が *maxmin-kind* であるとき、*reduction-spec* 内の *reduction-variable* は *scalar-variable-name* でなければならない。

制約: *reduction-variable* に指定された変数の型は、*reduction-kind* のおのおのに対して以下のものでなければならない。

`.AND.`, `.OR.`, `.EQV.`, `NEQV.` に対しては論理型。

`IAND`, `IOR`, `IEOR` に対しては整数型。

`+`, `*`に対しては数値型。

`MAX`, `MIN`, `FIRSTMAX`, `FIRSTMIN`, `LASTMAX`, `LASTMIN` に対しては整数型か実数型。

制約: *reduction-kind* なしの *reduction-clause* 内で指定された *reduction-variable* は、ループ内で HPF2.0 仕様書 5.1.3 項に規定される集計文の形式で参照されなければならない。(*reduction-kind* 付きの *reduction-clause* 内で指定された *reduction-variable* は、ループ内で任意の形式で参照されてよい。)

制約: *reduction-variable* または *location-variable* として現れる変数は、同じ *independent-directive* の中で複数回現れてはならず、*independent-directive* が適用される後続の *do-stmt*、*forall-stmt* および *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* および *reduction-clause* に現れてはならない。

A.4 通信最適化に関する HPF/JA 拡張

A.4.1 非同期転送機能

J401 *asyncid-directive* **is** `ASYNCID` *async-id-list*

J402 *async-id* **is** *async-id-name*

制約: `SAVE` が *combined-directive* に現れるときには、必ず `ASYNCID` も現れなければならない。

J403 *asynchronous-directive* **is** `ASYNCHRONOUS` *asynchronous-stuff*

J404 *asynchronous-stuff* **is** `(` [`ID =`] *async-id* `)` [`,` *nobuffer-clause*]

J405 *asynchronous-construct* **is**
hpfja-directive-origin block-asynchronous-directive
block
hpfja-directive-origin end-asynchronous-directive

J406 *block-asynchronous-directive* **is** `ASYNCHRONOUS` *asynchronous-stuff* `BEGIN`

J407 *end-asynchronous-directive* **is** `END ASYNCHRONOUS`

J408 *asyncwait-directive* **is** `ASYNCWAIT (` [`ID =`] *async-id* `)`

J409 *nobuffer-clause* **is** `NOBUFFER`

J410 *redistribute-directive-ja* **is** [*async-prefix*] *redistribute-directive*

J411 *realign-directive-ja* **is** [*async-prefix*] *realign-directive*

J412 *async-prefix* **is** `ASYNC` ([`ID =`] *async-id*)

A.4.2 SHADOW 指示文の拡張

J413 *shadow-spec-ja* **is** *width*
or *low-width* : *high-width*
or *full-width*

J414 *full-width* **is** `*`

制約: *shadow-spec-ja-list* の長さは、*shadow-target* の次元数と等しくなければならない。

制約: *shadow-spec-ja* として *full-width* を指定した場合、すべての次元で *full-width* を指定しなければならない。

A.4.4 REFLECT 指示文

J415 *reflect-directive* **is** [*async-prefix*] `REFLECT` *reflect-object-list*

J416 *reflect-object* **is** *object-name*

制約: *reflect-object* のデータ実体またはシャドウ実体を配置するプロセッサは、すべて活動プロセッサでなければならない。

A.4.5 ON 指示文の HOME 節の拡張

J417 *home-ja* **is** `HOME (` *variable*)
or `HOME (` *template-elmt*)
or `EXT_HOME (` *variable* [, *shadow-attr-stuff*])
or (*processors-elmt*)

制約: `EXT_HOME` 節の *shadow-attr-stuff* で指定される *shadow-spec-list* の長さは、*variable* の親実体の次元数と一致していなければならない。

制約: `EXT_HOME` 節の *shadow-attr-stuff* で指定される各次元の上下のシャドウ幅は、それぞれ *variable* の親実体の持つシャドウ幅と比較して等しいか小さくなければならない。

制約: *shadow-attr-stuff* の中の *shadow-spec-ja* は、*full-width*(アスタリスク)であってはならない。

A.4.6 LOCAL 節と LOCAL 指示文

J418 *on-stuff-ja* **is** *home-ja* [, *on-optional-clause-list*]

J419 *on-optional-clause* **is** *resident-clause*
or *local-clause*
or *new-clause*

J420 *local-clause* **is** `LOCAL` *local-stuff*

J421 *local-stuff* **is** [(*local-object-list*)]

J422 *local-directive* **is** `LOCAL` *local-stuff*

J423 *local-construct* **is**
hpfja-directive-origin block-local-directive
block
hpfja-directive-origin end-local-directive

J424 *block-local-directive* **is** `LOCAL` *local-stuff* `BEGIN`

J425 *end-local-directive* **is** `END LOCAL`

J426 *local-object* **is** *object*

A.4.7 通信スケジュール再利用

J427 *index-reuse-directive* **is** `INDEX_REUSE` [(*scalar-logical-expr*)]
index-reuse-variable-list

J428 *index-reuse-variable* **is** *array-variable-name*

附属書 B　構文記号の索引

この附属書は、構文規則で用いられる記号の索引を示す。J で始まる識別番号は、本 HPF/JA 言語仕様書の構文規則を表し、その規則の全体は附属書 A に示されている。H で始まる識別番号は、High Performance Fortran 言語仕様書の構文規則を表し、R で始まる識別番号は、Fortran 言語規格 (“Fortran 95”) の構文規則を表す。

B.1　構文規則の左辺に現れる非終端記号

記号	定義箇所	参照箇所
action-stmt	R216	J205
align-directive	H313	J203
async-id	J402	J401 J404 J408 J412
async-prefix	J412	J410 J411 J415
asynchronous-construct	J405	J205
asynchronous-directive	J403	J204
asynchronous-stuff	J404	J403 J406
asyncid-directive	J401	J203
asyncwait-directive	J408	J204
block	R801	J405 J423
block-asynchronous-directive	J406	J405
block-local-directive	J424	J423
case-construct	R808	J205
combined-directive	H301	J203
distribute-directive	H305	J203
do-construct	R816	J205
dynamic-directive	H804	J203
end-asynchronous-directive	J407	J405
end-local-directive	J425	J423
executable-construct-ja	J205	
executable-directive-ja	J204	
full-width	J414	J413
high-width	H823	J413
home	H907	
home-ja	J417	J418
hpf-directive	H203	J201
hpfja-directive-line	J201	

shadow-spec-ja	J413	
specification-directive-ja	J203	
subset-directive	H901	J203
task-region-construct	H917	J205
template-directive	H331	J203
template-elmt	H908	J417
variable	R601	J417
where-construct	R739	J205
width	H821	J413

B.2 構文規則の左辺に現れない非終端記号

記号	参照箇所
array-variable-name	J428
async-id-name	J402
object	J426
object-name	J416
variable-name	J307

B.3 終端記号

記号	参照箇所
!HPFJ	J202
(	J302 J404 J408 J412 J417 J421 J427
)	J302 J404 J408 J412 J417 J421 J427
*	J304 J414
*HPFJ	J202
+	J304
,	J301 J404 J417 J418
.AND.	J304
.EQV.	J304
.NEQV.	J304
.OR.	J304
/	J306
:	J302 J413
=	J404 J408 J412
ASYNC	J412
ASYNCHRONOUS	J403 J406 J407