High Performance Fortran
言語仕様書

High Performance Fortran Forum

January 31, 1997
Version 2.0
日本語訳第 1.0 版
1998 年 5 月 21 日

The High Performance Fortran Forum (HPFF), with participation from over 40 organizations, met from March 1992 to March 1993 to define a set of extensions to Fortran called High Performance Fortran (HPF). Our goal was to address the problems of writing data parallel programs for architectures where the distribution of data impacts performance. While we hope that the HPF extensions will become widely available, HPFF is not sanctioned or supported by any official standards organization. The HPFF had a second series of meetings from April 1994 to October 1994 to consider requests for corrections, clarifications, and interpretations to the Version 1.0 HPF document and also to develop user requirements for possible future changes to HPF. A third set of meetings took place From January 1995 through December 1996 to incorporate features recommended to meet user needs identified in the 1994 meetings.

This document contains all the technical features proposed for the version of the language known as HPF Version 2.0. This copy of the draft was processed by LaTeX on May 25, 1998.

HPFF encourages requests for interpretation of this document, and comments on the language defined here. We will give our best effort to answering interpretation questions, and general comments will be considered in future HPFF language specifications.

Please send interpretation requests to `hpff-interpret@cs.rice.edu`. Your request is archived and forwarded to a group of HPFF committee members who attempt to respond to it.

The text of interpretation requests becomes the property of Rice University.

© 1994, 1995, 1996, 1997 Rice University, Houston, Texas. Permission to copy without fee all or part of this material is granted, provided that the Rice University copyright notice and the title of this document appear, and notice is given that copying is by permission of Rice University.

© 1996, 1997, 1998 Fujitsu Limited, Hitachi, Ltd., and NEC Corporation, Japan.

日本語訳への注意書

1. 本マニュアルは、財団法人高度情報科学技術研究機構、三好副理事長の発案により組織された、HPF(High Performance Fortran) に関わる日本国内メーカ 3 社のコンパイラ開発者およびアプリケーション研究者・開発者から構成される JAHPF (Japan Association for HPF) の発意により作成された、HPF Version2.0 の日本語訳である。
2. 本マニュアルは、富士通株式会社・株式会社日立製作所・日本電気株式会社の共同著作物である。
3. 本マニュアルは、原文が Rice 大学の Copyright であり、日本語訳にも適用される。
4. 本マニュアルの複製、配布は、本著作権表示を含む限り自由である。ただし、商用で利益が発生する場合は、事前に上記 3 社との協議を必要とする。

目 次

第 I 部　序論 **1**

第 1 章　概要 **3**
1.1 High Performance Fortran の目的と適用範囲 3
1.2 HPF 言語モデル 4
1.2.1 データマッピング指示文 5
1.3 HPF 2.0 言語機能の概要 7
1.3.1 HPF 2.0 言語機能 7
1.3.2 HPF 2.0 公認拡張仕様 8
1.4 HPF 1.1 からの変更点 10
1.4.1 言語の分割 10
1.4.2 Fortran 規格に採用された機能 11
1.4.3 HPF 2.0 で除去または制約された機能 11
1.4.4 公認拡張仕様に移された機能 11
1.4.5 HPF 2.0 の新機能 11
1.4.6 新公認拡張仕様 11
1.4.7 外部機関で保持される認知された HPF 外来仕様 12

第 2 章　記述法と構文 **13**
2.1 記述法 13
2.2 指示文の構文 14

第 II 部　High Performance Fortran 言語 **17**

第 3 章　データマッピング **19**
3.1 モデル 19
3.2 データ整列とデータ分散指示文の構文 21
3.3 DISTRIBUTE 指示文 23
3.4 ALIGN 指示文 29
3.5 割付け配列とポインタ 36
3.6 PROCESSORS 指示文 38
3.7 TEMPLATE 指示文 41
3.8 記憶列結合と順序結合 43
3.8.1 記憶列結合 43

3.8.2 SEQUENCE 指示文 45

第 4 章 副プログラム境界でのデータマッピング 47
4.1 はじめに 47
4.2 どのような再マッピングが必要とされ、誰がそれを行なうのか 49
4.3 分散とプロセッサ構成 50
4.3.1 例 50
4.3.2 省略時の解釈 51
4.4 整列 52
4.4.1 仮引数のテンプレート 52
4.4.2 INHERIT 指示文 52
4.4.3 記述的な ALIGN 指示文 55
4.5 マッピングの集合における同値関係と半順序 57
4.6 明示的引用仕様を省略するための条件 59
4.7 手続の特性 60
4.8 引数の受け渡しと順序結合 61
4.8.1 順序結合規則 61
4.8.2 順序結合についての説明 61
4.8.3 順序結合の例 62

第 5 章 INDEPENDENT 指示文及び関連の指示文 63
5.1 INDEPENDENT 指示文 64
5.1.1 INDEPENDENT 指示文の視覚化 69
5.1.2 NEW 変数 71
5.1.3 REDUCTION 変数と REDUCTION 文 73
5.1.4 集計の実装と意味論 75
5.2 INDEPENDENT の進んだ例 78

第 6 章 外来プログラム単位 81
6.1 概要 81
6.2 外来プログラム単位の宣言 82
6.2.1 FUNCTION 文及び SUBROUTINE 文 82
6.2.2 PROGRAM 文、MODULE 文、及び BLOCK DATA 文 83
6.2.3 外来プレフィックス 83
6.3 HPF 外来副プログラムの呼出し 86
6.3.1 型、手続、データに対するアクセス 86
6.3.2 呼出しの効果 87
6.4 外来手続の例 89

第 7 章 組込み手続とライブラリ手続 91
7.1 表記法 91
7.2 システム問合せ組込み関数 91
7.3 計算組込み関数 92

7.4 ライブラリ手続 . . . 92
7.4.1 マッピング問合せサブルーチン . . . 92
7.4.2 ビット操作関数 . . . 93
7.4.3 配列集計関数 . . . 93
7.4.4 配列集計拡散関数 . . . 93
7.4.5 配列累計関数 . . . 95
7.4.6 配列ソート関数 . . . 100
7.5 組込み手続とライブラリ手続の総称名 . . . 100
7.5.1 システム 問合せ組込み関数 . . . 100
7.5.2 マッピング問合せサブルーチン . . . 100
7.5.3 ビット操作関数 . . . 100
7.5.4 配列集計関数 . . . 101
7.5.5 配列集計拡散関数 . . . 101
7.5.6 配列累計関数 . . . 101
7.5.7 配列ソート関数 . . . 103
7.6 組込み手続の仕様 . . . 103
7.7 ライブラリ手続の仕様 . . . 104

第 III 部 HPF 公認拡張仕様 143

第 8 章 データマッピングの公認拡張仕様 145
8.1 拡張されたモデル . . . 145
8.2 拡張データマッピング指示文の属性形式の構文 . . . 146
8.3 REDISTRIBUTE 指示文 . . . 147
8.4 REALIGN 指示文 . . . 148
8.5 DYNAMIC 指示文 . . . 148
8.6 再マッピングと副プログラム・インタフェース . . . 150
8.7 部分プロセッサへのマッピング . . . 151
8.8 ポインタ . . . 153
8.8.1 マップされたポインタ . . . 153
8.8.2 ポインタと副プログラム . . . 155
8.8.3 ポインタや指示先に関する制限事項 . . . 156
8.9 構造体成分のマッピング . . . 157
8.10 新しい分散形式 . . . 162
8.11 RANGE 指示文 . . . 165
8.12 SHADOW 指示文 . . . 167
8.13 マッピングの集合における同値と半順序 . . . 168
8.14 明示的引用仕様を省略できる条件 . . . 169
8.15 手続の特性 . . . 170

第 9 章　データとタスク並列に対する公認拡張　171
9.1 活動プロセッサ集合 . . . 171
9.1.1 SUBSET 指示文 . . . 173
9.1.2 局所的な実体と仮引数のマッピング . . . 174
9.1.3 活動プロセッサに関する他の制限 . . . 175
9.2 ON 指示文 . . . 178
9.2.1 ON 指示文の文法 . . . 178
9.2.2 ON 指示文の意味 . . . 180
9.2.3 ON 指示文の例 . . . 184
9.2.4 副プログラムの実行に対する ON 指示文 . . . 190
9.3 RESIDENT 節、指示文、構文 . . . 195
9.3.1 RESIDENT 節の例 . . . 200
9.3.2 RESIDENT 指示文の手続引用への適用 . . . 203
9.4 TASK_REGION 構文 . . . 206
9.4.1 TASK_REGION 構造の文法 . . . 206
9.4.2 TASK_REGION 構文の意味 . . . 206
9.4.3 実行モデルと使用法 . . . 207
9.4.4 実装 . . . 208
9.4.5 例: 2-D FFT . . . 209

第 10 章　非同期入出力に関する公認拡張　213
10.1 WAIT 文 . . . 215

第 11 章　HPF 外来機能に関する公認拡張　217
11.1 様々な外来モデル：ローカルモデルとシリアルモデル . . . 217
11.1.1 ローカルモデル . . . 218
11.1.2 シリアルモデル . . . 220
11.2 外来言語との結合仕様 . . . 221
11.2.1 引数の制御 . . . 221
11.3 HPF 言語との結合仕様 . . . 224
11.3.1 HPF_LOCAL に関する特記事項 . . . 224
11.3.2 引数の結合 . . . 225
11.3.3 HPF_SERIAL に関する特記事項 . . . 227
11.4 C 言語との結合仕様 . . . 229
11.4.1 C 言語で定義された手続に対する引用仕様 . . . 229
11.4.2 C 言語に対するデータ型変換の仕様 . . . 230
11.5 Fortran 言語との結合仕様 . . . 234
11.6 Fortran 77 言語との結合仕様 . . . 234
11.6.1 F77_LOCAL に関する特記事項 . . . 235
11.6.2 F77_LOCAL 手続への引数の渡し方 . . . 235
11.6.3 F77_LOCAL 手続のプログラミング例 . . . 237
11.7 外来手続のためのライブラリ . . . 242
11.7.1 ローカル HPF 手続のためのライブラリ . . . 242

11.7.2 シリアル外来種別からのライブラリ引用 251

第 12 章 HPF 組込み手続とライブラリ手続の公認拡張 253
12.1 拡張組込み手続きの仕様 254
12.2 拡張ライブラリ手続きの仕様 256

第 IV 部 附属書 265

附属書 A 構文規則 267
A.2 記述法と構文 267
A.2.2 指示文の構文 267
A.3 データマッピング 268
A.3.2 データ整列とデータ分散指示文の構文 268
A.3.3 DISTRIBUTE 指示文 269
A.3.4 ALIGN 指示文 270
A.3.6 PROCESSORS 指示文 272
A.3.7 TEMPLATE 指示文 272
A.3.8 記憶列結合と順序結合 272
A.4 副プログラム境界でのデータマッピング 272
A.4.4 整列 272
A.5 INDEPENDENT 指示文及び関連の指示文 272
A.5.1 INDEPENDENT 指示文 272
A.6 外来プログラム単位 274
A.6.2 外来プログラム単位の宣言 274
A.8 データマッピングの公認拡張仕様 277
A.8.2 拡張データマッピング指示文の属性形式の構文 277
A.8.3 REDISTRIBUTE 指示文 277
A.8.4 REALIGN 指示文 278
A.8.5 DYNAMIC 指示文 278
A.8.7 部分プロセッサへのマッピング 278
A.8.9 構造体成分のマッピング 279
A.8.10 新しい分散形式 280
A.8.11 RANGE 指示文 281
A.8.12 SHADOW 指示文 281
A.9 データとタスク並列に対する公認拡張 282
A.9.1 活動プロセッサ集合 282
A.9.2 ON 指示文 282
A.9.3 RESIDENT 節、指示文、構文 282
A.9.4 TASK_REGION 構文 283
A.10 非同期入出力に関する公認拡張 283
A.10.1 WAIT 文 283
A.11 HPF 外来機能に関する公認拡張 284

A.11.2 外来言語との結合仕様 284

附属書 B 構文記号の索引 285
B.1 構文規則の左辺に現れる非終端記号 285
B.2 構文規則の左辺に現れない非終端記号 289
B.3 終端記号 290

附属書 C HPF 1.1 Subset 293
C.1 Fortran 90 Features in the HPF 1.1 Subset 293
C.2 HPF 1.1 Directives and Language Extensions in the HPF 1.1 Subset . . . 295

附属書 D Previous HPFF Acknowledgments 297
D.1 HPFF Acknowledgments 297
D.2 HPFF94 Acknowledgments 300

附属書 E Policy and Mechanism for Recognized Extrinsic Interfaces 305
E.1 Extrinsic Policy 305
E.2 Extrinsic Interface Mechanism 306

附属書 F HPF_CRAFT 307
F.1 Introduction 307
F.2 Examples of Use 308
F.3 External Interface 311
F.4 Execution Model 312
F.5 HPF_CRAFT Functional Summary 312
F.5.1 Data Mapping Features 313
F.5.2 Subprogram Interfaces 313
F.5.3 The INDEPENDENT Directive 314
F.5.4 The ON Clause 314
F.5.5 Array Syntax 315
F.5.6 Treatment of FORALL and WHERE Statements 316
F.5.7 Synchronization Primitives 316
F.5.8 Barrier Removal 317
F.5.9 Serial Regions 317
F.5.10 Libraries 318
F.5.11 Parallel Inquiry Intrinsics 318
F.5.12 Task Identity 318
F.5.13 Parallelism Specification Directives 318
F.5.14 The SYMMETRIC Directive 319
F.5.15 The RESIDENT Directive 319
F.5.16 The ATOMIC UPDATE Directive 319
F.5.17 The GEOMETRY Directive 320

附属書 G The FORTRAN 77 Local Library **321**
G.1 Introduction 321
G.2 Summary 322
G.3 Global HPF Subgrid Inquiry Routine 323
G.3.1 Subgrid Inquiries Involving Embedding Arrays 325
G.4 Local FORTRAN 77 Inquiry Routines 325
G.5 Programming Example Using HPF_SUBGRID_INFO 331
G.5.1 HPF Caller 331
G.5.2 FORTRAN 77 Callee 332
G.6 Programming Example Using F77-Callable Inquiry Subroutines 333
G.6.1 HPF Caller 333
G.6.2 FORTRAN 77 Callee 334

Acknowledgments

本ドキュメントは、富士通株式会社、株式会社日立製作所、日本電気株式会社の有志によって High Performance Fortran 2.0 言語仕様 (HPF2.0) を和訳したものである。和訳の作業は JAHPF(Japan Association for High Perforamance Fortran) の活動の一環として行なわれた。

HPF2.0 は、High Performance Fortran Forum (HPFF) にてまとめられた仕様であり、種々の並列計算機上で共通して利用可能な、並列処理のための言語仕様である。主に分散メモリ型のシステム上で簡潔に効率の良い並列処理を記述することを目的として設計されている。

JAHPF は、HPF 言語の実用性の向上および普及を目的として 1997 年に結成された検討会であり、40 人を超える日本の高性能コンピューティング専門家のグループで、主にスーパーコンピュータユーザおよびコンパイラベンダから構成される。

HPF2.0 の和訳作業の担当者は以下の通りである (敬称略)。

- 岡田 信、岩下英俊、末安 直樹 (富士通株式会社)
- 小林篤、根岸清、西谷康仁、太田寛 (株式会社日立製作所)
- 林康晴、末広謙二、村井均、妹尾義樹 (日本電気株式会社)

和訳に際しては、技術用語の訳語の選定と、それらの訳文への統一的な適用に細心の注意が払われた。訳語の選定は、JAHPF WG1 のメンバにより行なわれた。また、JAHPF の会員諸氏には、訳文のチェックを始めとして、本ドキュメントの品質向上に種々の助力をいただいた。この場を借りてお礼申し上げる。

最後に、JAHPF の活動や、本ドキュメントの作成に際し、広範に渡りご支援をいただいた (財) 高度情報科学技術研究機構 (RIST) に謝意を表する。特に、RIST の秦万美子、高橋由香のお二人には会議開催のサポートなど、いろいろとお世話をいただいた。

以下に本ドキュメントの元となった HPF2.0 仕様書の Acknowledgement をそのまま添付する。

The High Performance Fortran Forum (HPFF) is a coalition of industrial and academic groups working to suggest a set of standard extensions to Fortran that provide support for high performance programming on a wide variety of machines, including massively parallel SIMD and MIMD systems and vector processors. From its beginning, HPFF has included most vendors delivering parallel machines, a number of government laboratories, and university research groups. Public input has been encouraged. This document defining HPF 2.0 is the third in a series of documents resulting from the HPFF. HPF 2.0 is intended to be a language portable from workstations to massively parallel supercomputers while being able to express the algorithms needed to achieve high performance on specific architectures. HPF 2.0 builds on the efforts of the previous HPFF meetings, primarily in 1992 and 1994.

Specific acknowledgments for the many people who contributed to the previous versions of HPF are included in Annex D.

HPFF 2 Acknowledgments

The HPF 2.0 version of the document was prepared during series of meetings in 1995–1996. A number of people shared technical responsibilities for the activities of these meetings:

- Ken Kennedy, Convener and Meeting Chair;
- Mary Zosel, Executive Director;
- Rob Schreiber, Organizer for Control Subgroup;
- Piyush Mehrotra and Guy Steele, Organizers for Distribution Subgroup;
- David Loveman, Organizer for External Subgroup;
- Chuck Koelbel, Editor, assisted by multiple committee members (names later).

Attendance at the HPFF 2 meetings included the following people from organizations that were represented two or more times.

John Levesque, Marc Baber ..... Applied Parallel Research
Ian Foster ..... Argonne National Laboratory
Jaspal Subhlok ..... Carnegie Mellon University
Jim Cowie ..... Cooperating Systems
Andy Meltzer ..... Cray Research
Henk Sipps, Will Denissen ..... Delft University of Technology
David Loveman, Carl Offner ..... Digital Equipment Corporation
Joel Williamson ..... Hewlett Packard/Convex
Henry Zongaro ..... IBM
Piyush Mehrotra ... Institute for Computer Applications in Science & Engineering
Mary Zosel ..... Lawrence Livermore National Laboratory
Bob Boland, Ralph Brickner, Joe Fasel ..... Los Alamos National Laboratory
J. Ramanujam, Arun Venkatachar ..... Louisiana State University
Steve Hammond ..... National Center for Atmospheric Research
Yoshiki Seo ..... NEC
Alok Choudhary ..... Northwestern University/Syracuse University
P. Sadayappan, ... Ohio State University
Larry Meadows, Zeki Bozkus ..... The Portland Group, Inc.
Robert Schreiber ..... RIACS and Hewlett Packard
Ken Kennedy, Charles Koelbel ..... Rice University
Hon Yau, Tom Haupt ..... Syracuse University
Guy Steele ..... Sun Microsystems
Paula Vaughn ..... Mississippi State University

Carol Munroe, Harvey Richardson ..... Thinking Machines Corporation
Scott Baden, Steve Fink, Jay Boisseau ..... University of California, San Diego
Robert Babb ..... University of Denver
Joel Saltz, Paul Havlak, Manuel Ujaldon ..... University of Maryland
Jerrold Wagener ..... X3J3 and University of Oklahoma
Barbara Chapman, Siegfried Benkner, Guy Robinson ..... University of Vienna

An important activity of HPFF 2 meetings was the processing of the many items submitted for comment, correction, and interpretation. General thanks go to all who submitted comments and interpretation requests. The following people submitted items that resulted in changes to the document text: Yasuharu Hayashi, Michael Hennecke, Larry Meadows, Carol Munroe, Eiji Nunohiro, Carl Offner, Rob Schreiber, David Singleton, Shigeru Uzuhara, Clint Whaley, and—last, but most prolific—Henry Zongaro.

Special thanks go to Chuck Koelbel at Rice University for continued maintenance of the HPFF mailing lists.

Theresa Chatman and staff at Rice University were responsible for meeting planning and organization and Danny Powell continued to handle financial details of the project.

HPFF 2 received direct support for research and administrative activities from grants from ARPA, DOE, and NSF.

第I部

序論

ここでは、本書の構成の全体について記述する。それは、High Performance Fortran 2.0 版 (第 II 部に記述される) と、HPF 公認拡張仕様 (第 III 部に記述される) に共通の用語と概念を定義し、以下の章のために必要な基礎を与える。

第1章　概要

本書は、High Performance Fortran (HPF) 言語で表現されたプログラムの形式を明示し、その解釈を規定する。HPF は確立された国際標準 Fortran 規格への拡張および変更の集合として設計されている。本書の公表時点では、基準として使われている標準は “Fortran 95” (ISO/IEC 1539:1997) と非公式に呼ばれている。その文書への参照は次のように行なわれる: その文書の 13.11.6 は、F95:13.11.6 として参照される。

本章では、言語の目的と適用範囲を概説し、HPF 言語モデルを紹介し、言語の主要機能を強調し、HPF 1.1 と HPF 2.0 の間の差異を説明し、そして本書の残りの部分の手引きを与える。

1.1　High Performance Fortran の目的と適用範囲

HPF 言語の開発の背景にある主要な目的は:

- データ並列プログラミング (単一スレッド、グローバルな名前空間、疎同期並列計算) のサポート
- 異なったアーキテクチャにまたがる可搬性
- 不均一なメモリアクセスコストをもった並列計算機上で高性能 (ただし、他の計算機上での性能も阻害しない)
- 基準として Fortran 規格 (現在は Fortran 95) を使用
- 他言語 (例えば、C) および他のプログラミングパラダイム (例えば、MPI を使用したメッセージ通信) とのオープンなインタフェースと相互引用性

二次的な目的は:

- 限られた時間内での実装可能性
- Fortran および C の将来の標準化活動への入力の提供
- 進んだ機能を一貫した方法で言語に付加するための発展的な道筋の提供

言語定義の最初の版 HPF 1.0 は 1993 年 5 月に公開された。HPF 1.0 に定義されていた多くの言語機能は今や Fortran 95 言語規格に吸収されている (例えば、単純 `FORALL` 文および構文、`PURE` 手続)。したがって、これらの機能はもはや HPF 2.0 の定義の中では詳細には記述されない。HPF 言語の進展 (1.0、 1.1 と 2.0 版を通して) についての情報と、HPF 2.0 と HPF 1.1 の相違点の列挙は 1.4 節にある。

1.2 HPF 言語モデル

HPF の重要な目的は、多様な並列計算機間にまたがるプログラムの可搬性を達成することである。そのためには、HPF プログラムがすべての計算機上で動作するだけではなく、ある並列計算機上で高い性能をもつ HPF プログラムは、他の並列計算機上でも相応のプロセッサ数で適度に高い性能を達成できなければならない。そうでなければ、ある計算機上で高い性能を達成するために費やされたプログラマの努力は、HPF プログラムを他の計算機に移植した場合に無駄になってしまう。共有メモリ計算機と分散メモリ計算機は異なる低レベル操作 (primitives) を使用するが、これらの計算機上での並列プログラムの性能に影響する基本的な要因に関しては広い類似性が存在する。したがって、ある程度高いレベルの HPF プログラムについて、異なる計算機上で高い効率を達成することは実現可能な目的である。並列プログラムの性能に影響する基本的な要因は、利用可能な並列度、データローカリティの利用、適当なタスク粒度の選択、などである。HPF はこれらの要因に関して、プログラマがコンパイラに指示する機構を提供する。

HPF の最初の版は Fortran 90 の拡張として定義されていた。HPF 2.0 は現在の Fortran 規格 (Fortran 95) の拡張として定義される。HPF の将来の改定版は、ISO で承認されしだい Fortran 規格の進展を包含し、それらと一貫性をもつ予定である。

Fortran を基準にして、HPF 言語機能は次の 4 つに分類できる:

- HPF 指示文
- 新言語構文
- 新ライブラリルーチン
- 言語の変更および制約

HPF 指示文は、コンパイラに対して実装戦略を示唆したり、プログラムについての事実を表明するための構造化された注釈として現れる。正しく使われた場合には、それらは実行される計算の効率にのみ影響を与え、プログラムによって計算される値は変えない。HPF 指示文の形式は、文の先頭にある注釈先頭語を削除することによって、将来の Fortran 規格がこれらの機能を完全な文として言語の中に包含することができるように選ばれている。

いくつかの新言語機能は Fortran 構文および解釈の直接の拡張として定義されている。新 HPF 言語機能は、第一種言語構造であり、プログラムによって計算される結果に直接に影響するという点において HPF 指示文とは異なっている。

計算機能をもつ HPF ライブラリは、高性能計算のために有用であると証明されているルーチンへの標準インタフェースを定義する。これらの追加関数には、マッピング問合せ、ビット操作、配列集計、配列集計拡散、配列累計、それに配列ソート関数がある。

Fortran 95 に対する少数の変更と制約が定義されている。最も重要な制約は、HPF のデータ分散機能と両立しないことから、順序結合と記憶列結合の使用に対して課せられた制約である。しかし、ある HPF 指示文に明示することによって、順序結合と記憶列結合を維持することも可能である。

1.2.1 データマッピング指示文

HPF における並列性の基本モデルは、グローバルな共用アドレス空間をもつ単一スレッドデータ並列実行モデルである。Fortran の配列記述と FORALL 文はデータ並列計算を指定する自然な方法である。加えて、HPF は INDEPENDENT 指示文を備えている。INDEPENDENT 指示文は、ループの繰返しをまたぐ依存関係がなく、したがって並列実行可能であることを表明している。

データローカリティの利用は、単一プロセッサのワークステーションであろうと、ワークステーションのネットワークであろうと、並列計算機であろうと、高性能計算機で高い性能を得るためには決定的に重要である。不均一メモリアクセス (NUMA) 並列計算機上では、プロセッサメモリ間に効果的にデータを分散することは、データ移動のオーバヘッドを減らすために非常に重要である。HPF の基本的な機能の一つはデータマッピングを利用者が指定できる機能である。HPF は、並列計算機を抽象プロセッサの一次元あるいはそれ以上の次元の矩形構成とみなす機能を備えている。プログラマは、異なる配列間で配列要素の相対的な整列および論理プロセッサ格子上への配列の分散を指定することができる。データマッピングは、並列性能を最適化するときにコンパイラの助けとなるが、プログラムの意味には影響を与えない HPF 指示文を使って指定される。以下の簡単な例で説明する。

```
      REAL A(1000,1000)
!HPF$ PROCESSORS procs(4,4)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO procs :: A
      DO k = 1, num_iter
         FORALL (i=2:999, j=2:999)
           A(i,j) = (A(i,j-1) + A(i-1,j) + A(i,j+1) + A(i+1,j))/4
         END FORALL
      END DO
```

このプログラムは、一つの二次元浮動小数点配列 A を使った単純な Jacobi 緩和法を記述したものである。HPF 指示文は構造化された注釈として現れている。PROCESSORS 指示文は 4×4 のプロセッサの論理格子 proc を指定している。DISTRIBUTE 指示文はコンパイラが配列 A をその各次元に沿って等しい大きさのブロックに分割することを推奨している。この結果、 250×250 の要素を含む 4×4 個のブロックができ、各プロセッサに 1 ブロックずつ割り振られる。PROCESSORS と DISTRIBUTE 指示文の詳細は、後の第 3 章で説明される。

外側の DO k ループは num_iter 回だけ Jacobi 緩和ステップを繰り返す。内側のループは Fortran 95 の FORALL 構文を使っている。それは、ループ本体を i と j に対して 2 から 999 までの範囲のすべての値について実行することを指定している。FORALL では、左辺の変数への代入が行われる以前にすべての繰返しについて (すなわち、i と j に対して 2 から 999 までの範囲のすべての値について) 右辺の式の評価が行われる必要がある。

このプログラムを 16 台のプロセッサからなる分散メモリ計算機上で実行する場合、HPF コンパイラは各プロセッサがグローバルな配列 A の一部をローカルに含むような SPMD コードを生成する。内側の FORALL は並列実行され外側の k ループは逐次実行される。各プロセッサは、他のプロセッサのローカルメモリにマップされている A の部分に含まれる「境界」の

要素を必要とする。HPF コンパイラは、必要なプロセッサ間通信を行う低レベル操作を、生成された SPMD コードに挿入する。グローバルな名前空間をもつ単一スレッドデータ並列モデルでは、プログラマは、並列化とデータ分割のための戦略をより高い抽象度で容易に指定できる。抽象的なグローバルな名前空間から個別のプロセッサのローカルメモリへの (アドレス) 変換や、明示的なプロセッサ間通信の管理などの退屈な低いレベルの処理の詳細はコンパイラに任されている。

次の例はスカラ代入文によって起こる暗黙の通信の例である。この例の目的は、並列実行での通信の必要性に対してデータの分散指定がどう関わっているかを示すことである。説明は必ずしも実際の翻訳の過程を反映しているわけではない。

以下のプログラムを考える:

```
      REAL a(1000), b(1000), c(1000), x(500), y(0:501)
      INTEGER inx(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b, inx
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: c
!HPF$ ALIGN x(i) WITH y(i+1)
      ...
      a(i) = b(i)                    ! 代入 1
      x(i) = y(i+1)                  ! 代入 2
      a(i) = c(i)                    ! 代入 3
      a(i) = a(i-1) + a(i) + a(i+1)  ! 代入 4
      c(i) = c(i-1) + c(i) + c(i+1)  ! 代入 5
      x(i) = y(i)                    ! 代入 6
      a(i) = a(inx(i)) + b(inx(i))   ! 代入 7
```

この例では、PROCESSORS 指示文は 10 個のプロセッサの一次元構成を指定している。DISTRIBUTE 指示文は、1 プロセッサ当たり 100 個の連続する要素から成るブロックを構成するように、配列 a、 b および inx を 10 個のプロセッサに分散することをコンパイラに推奨する。配列 c は、procs(1) には c(1), c(11), ..., c(991) 、procs(2) には c(2), c(12), ..., c(992) 、などとなるように各プロセッサに循環的に分散される。配列 x と y のプロセッサへの完全なマッピングは指定されていない。しかし、それらの相対的な整列は、ALIGN 指示文によって示されている。ALIGN 指示文は、コンパイラによって選ばれる y の実際の分散にかかわらず、x(i) と y(i+1) はすべての i の値に対して同じプロセッサに置かれるよう推奨している (y(0) と y(1) は x のどの要素とも整列していない)。PROCESSORS、DISTRIBUTE、および ALIGN 指示文の詳細は第 3 章で説明される。

代入 1 (a(i) = b(i)) について、a と b が同一の分散であることは、すべての i について a(i) と b(i) が同じプロセッサにマップされるべきであることを指定している。したがって、この文の実行ではデータ値のプロセッサ間通信は必要でない。

代入 2 (x(i) = y(i+1)) では本質的に通信は必要ない。この場合、配列のどのような分散に対しても二つの配列の相対的な整列は代入文と一致している。

代入 3 (a(i) = c(i)) は最初の代入と非常に似ているように見えるが、a と c の分散が

異なっているため、通信の必要性についてはまったく異なっている. 配列要素 a(i) と c(i) は i の可能な値のたった 10%の場合しか同一プロセッサにマップされていない。(このことは第 3 章にある BLOCK と CYCLIC の定義からわかる。) 各要素は $\lfloor (\mathtt{i}-1)/100 \rfloor = (\mathtt{i}-1) \bmod 10$ が成立する場合のみ同じプロセッサに置かれている。例えば、$\mathtt{i} = 1$ または $\mathtt{i} = 102$ の場合にはこの代入は本来の通信を含まない (すなわち、a(i) と c(i) の両方とも同じプロセッサにある)。しかし、$\mathtt{i} = 2$ の場合には通信が必要である。

代入 4 (a(i) = a(i-1) + a(i) + a(i+1)) では、配列 a への参照は i の可能な値の 98%の場合に同じプロセッサ上で行われる。これの例外は、$k = 1, 2, \ldots, 9$ に対して $\mathtt{i} = 100 * \mathtt{k}$ となる場合 (a(i) と a(i-1) が procs(k) 上にあり、a(i+1) が procs(k+1) にある場合) と、$k = 1, 2, \ldots, 9$ に対して $\mathtt{i} = 100 * \mathtt{k} + 1$ となる場合 (a(i) と a(i+1) が procs(k+1) 上にあり、a(i-1) が procs(k) にある場合) である。この代入文では各プロセッサ上の「境界」要素についてのみ通信が必要である。

代入 5 (c(i) = c(i-1) + c(i) + c(i+1)) は、表面的には代入 4 と同じに見えるがまったく異なった通信動作をする。c の分散が BLOCK ではなく CYCLIC であることから、c(i)、c(i-1) および c(i+1) の三つはすべての i の値に対して三つの異なったプロセッサにマップされている。したがって、実装方法にかかわらず、この文は右辺の参照のために少なくとも 2 回の通信が必要である。

最後の二つの代入は通信の必要性に関して非常に限られた情報しかもっていない。代入 6 (x(i) = y(i)) では使用可能な情報は x(i) と y(i+1) が同じプロセッサにあるということだけである。これからは x(i) と y(i) の関係についての論理的な帰結は導けない。したがって、さらなる情報が無ければ、この文のために実行時に必要な通信に関しては何も言うことができない。代入 7 (a(i) = a(inx(i)) + b(inx(i))) では、a(inx(i)) と b(inx(i)) が常に同じプロセッサ上にマップされていることが証明できる。同様に、a(i) と inx(i) が一緒にマップされることを簡単に推論できる。しかし、inx に格納されている値の知識なしには a(i) と a(inx(i)) の関係は分からない。同様に、a(i) と b(inx(i)) の関係も分からない。

1.3 HPF 2.0 言語機能の概要

本書で定義される言語は、二つの主要部分から成る:

- HPF 2.0 言語 (第 II 部)
- HPF 2.0 公認拡張仕様 (第 III 部)

HPF 2.0 言語は言語仕様が公開されてから 1 年以内に実装可能と思われる機能を含んでいる。これらは、基本データ分散機能、データ並列機能、組込みおよびライブラリルーチン、それに外来機構を含んでいる。公認拡張仕様は、特定の要求を満足するが、初期のコンパイラの実装ではサポートされそうにない進んだ機能を含んでいる。

1.3.1 HPF 2.0 言語機能

1.3.1.1 データ分散機能 (第 3 章 と 第 4 章)

大抵の並列アーキテクチャおよび逐次アーキテクチャでは、データのアクセスにローカリティがある場合に最も高い性能を発揮する。Fortran 規格に暗黙的に含まれている記憶順序の逐

次性は、アーキテクチャが要求するローカリティとしばしば相反する。これを避けるために、HPF では、データの相互位置付け (ALIGN) と、メモリ域または抽象プロセッサ間でのデータの分割 (DISTRIBUTE) を記述する機能を含んでいる。コンパイラは、すべてのデータはプログラム内の任意の時点で単一の値をもつという意味的な制約に従う限り、データのメモリ割付けを改善するためにこれらの記述を利用することができる。第 4 章では、マッピング機能が副プログラム境界を越えてどのように作用するかを定義する。

HPF の目的の一つは Fortran と互換性を保つことであるが、Fortran の順序結合と記憶列結合を完全に維持することは、HPF の中でデータの分散を通して高性能を得るという目的と両立しない。第 3 章 と 第 4 章では記憶列結合と順序結合に関係する制約と指示文を説明する。

1.3.1.2 データ並列実行機能 (第 5 章)

並列計算を明示的に表現するために、 HPF は INDEPENDENT 指示文を定義している。INDEPENDENT 指示文はプログラムの特定部分にある文が逐次的依存性を示さないことを表明する。それは、正しく使われた場合にはその部分の意味を変えず、最適化のためにより多くの情報を言語処理系に与える可能性がある。INDEPENDENT 指示文の REDUCTION 節は、交換則と結合則が成立する演算によって更新される変数を識別するために使用される。REDUCTION 節は、変数の更新の累算が順序に依存しないようなループの場合に、集計演算がもつ並列性を利用可能にする。

1.3.1.3 外来プログラム単位 (第 6 章)

HPF は高レベル機種独立言語として設計されていることから、直接表現するのが困難または不可能な操作が存在する。例えば、あるアプリケーションではある機種上で高度にチューニングされたシストリック通信が役に立つかもしれない。HPF のグローバルなアドレス空間ではこれをうまく表現できない。HPF では、外来機構を使って、明示的なメッセージ通信サブルーチンライブラリや他の言語 (例えば C) での記述のように、他のパラダイムで記述された手続と容易にインタフェースを取ることができる。

1.3.1.4 組込み関数と標準ライブラリ (第 7 章)

大規模並列計算機での経験から、並列アルゴリズムの設計に有用な多くの基本操作が知られている。Fortran の組込み配列関数はこれらのいくつかに応えている。HPF では、組込み関数および標準ライブラリ関数としていくつかの並列操作を言語定義に追加している。さらに、並列実行の制御に有用ないくつかのシステム問合せ関数も備えられている。

1.3.2 HPF 2.0 公認拡張仕様

1.3.2.1 データマッピングの拡張機能 (第 8 章)

拡張されたマッピング機能は、実行時におけるデータの動的な再整列と再分散 (REALIGN、REDISTRIBUTE、 DYMNAMIC 指示文)、プロセッサの部分集合へのデータマッピング、ポインタと構造型成分のマッピング、およびデータの不規則分散のサポート (GEN_BLOCK、 INDIRECT

分散) を包含し、データマッピングをより自由に制御することを可能にしている。さらに、配列が取り得る可能な分散の範囲の情報 (RANGE 指示文) や、ステンシルを使った隣接計算に含まれる配列で使用されるバッファの大きさの情報 (SHADOW) を、プログラマがコンパイラに与えることができる機構が定義されている。

1.3.2.2 データ並列とタスク並列の拡張機能 (第 9 章)

ON 指示文によって計算の分割を明示的に指定できる。計算を実行するように推奨するプロセッサは、プロセッサ構成の明示的に指定された部分集合として、または間接的に、データ実体またはテンプレートがマップされているプロセッサの集合として、のどちらかで指定できる。

コンパイラが効率的なコードを生成するのを助けるために、プログラマによって ON 指示文と共に使われる RESIDENT 指示文が定義されている。RESIDENT 指示文は、ON 指示文の有効範囲内の指定された実体へのすべてのアクセスが実行中のプロセッサにローカルに閉じることを表明するために使用される。TASK_REGION 指示文は、重なりのない部分プロセッサの間でプログラムの異なった部分を同時実行することを利用者が指定できるようにする。

1.3.2.3 非同期入出力の拡張機能 (第 10 章)

入出力と計算の同時実行を可能にするために、書式なしデータに対する非同期直接READ/WRITE のための拡張が定義されている。非同期入出力のために、Fortran の READ/WRITE 文に非ブロッキング実行を指定する入出力制御パラメタが追加され、新しい文 (WAIT) が導入されている。

1.3.2.4 組込みおよびライブラリ手続の拡張 (第 12 章)

HPF 組込み関数およびライブラリルーチンに対する公認拡張仕様は、ほとんどがマッピング問合せ手続に関係している。いくつかの新しい問合せルーチンが定義されている。HPF 2.0 言語で定義されている他のルーチンは、部分プロセッサへのマッピング、GEN_BLOCK、 INDIRECT、DYMNAMIC 分散などの拡張マッピング機能に関する問合せを行えるように拡張されている。Fortran の TRANSPOSE 組込み関数の汎用化も定義されている。

1.3.2.5 HPF 外来仕様の公認拡張仕様 (第 11 章)

第 11 章には、公認 HPF 2.0 拡張機能として多くの特定の外来インタフェースが定義されている。これらは、異なった並列処理モデルへのインタフェース (SPMD 並列処理のための LOCAL や単一プロセスでの逐次実行のための SERIAL) はもちろん、他言語 (例えば、C および FORTRAN 77) との相互引用性を容易にするインタフェースも含んでいる。外来機能において有用なライブラリルーチンは 11.7 節に定義されている。HPF フォーラムによって公式に認知されているがフォーラムによって定義または保守されていない付加的な外来インタフェースは、附属書 F と G に含まれている。このような外来インタフェースの公式認知のための方針と方法は附属書 E に記述されている。

1.4 HPF 1.1 からの変更点

HPF 2.0 は、様々な点で HPF 1.1 と異なっている:

言語の再分割: 新しい文書は、HPF 2.0 と公認拡張仕様の二つの構成要素を記述している。HPF 2.0 言語は、幅広く比較的迅速に実装されることが期待される。公認拡張仕様は、HPF 2.0 の一部ではないが、利用者の要求に応じて、コンパイラ技術の成熟に伴って、将来の実装に含まれるかも知れない。

Fortran 規格に採用された機能: Fortran 90 ではなく Fortran が拡張のための基準言語として定義される。このことは、HPF は 1995 年改訂版 Fortran に追加されたすべての機能を含むことを意味する。この改訂版では HPF 1.1 のいくつかの機能は Fortran 規格の一部になった。したがって、Fortran に対する HPF 拡張としては、もはや現れない。

HPF 2.0 で除去または制約された機能: 現在までに実装されなかった HPF 1.1 のいくつかの機能は、除去することによって得られる単純さが機能の有効性を上回ることが経験により示されたため、言語から除去された。

HPF サブセットの削除: HPF 1.1 とは違って、HPF 2.0 はより早期の実装を目的とした推奨最小サブセット (すなわち、HPF サブセット) をもたない。ただし、元の HPF 1.1 サブセットは附属書に加えておく。

公認拡張仕様に移された機能: いくつかの言語機能は HPF 1.1 から公認拡張仕様の分類に移された。

HPF 2.0 の新機能: いくつかの新機能が基準言語に追加された。

新公認拡張仕様: 多くの最新機能が言語への公認拡張仕様として定義された。

外部機関で保持される認知された HPF 外来仕様: 最後に、本書では、HPF 関連 `EXTRINSIC` インタフェースという新しい範疇を認めている。そしてそれは、そのようなインタフェースのための適当な基準を満たすと承認されたが、公認拡張仕様には含まれない。そのようなインタフェースの内容の責任は、HPF フォーラムではなくそれを提案した機関にあるとされている。

これらの項目の各々は以下の節にまとめられている。

1.4.1 言語の分割

HPF フォーラムはしばしば相反する二つの重要な目的をもっていた:

- 利用者が要求する進んだ言語能力を供給すること。
- ベンダーによるコンパイラの早期開発を考慮すること。

両方の目的を満足するために、我々は言語定義を二つの部分に分けるという妥協を行った。HPF 2.0 は HPF 1.1 に非常によく似ており、本書が出てからだいたい 1 年以内に多くのベンダーによって効率的に実装されることが期待されている。実装のためにさらに多くの努力が

必要な進んだ機能は公認拡張仕様として集められている。実装者はできる限り迅速にこれらの機能をサポートするよう奨励される。また、利用者はかれらの要請をベンダーに知らせることによって、この過程を促進するよう奨励される。

1.4.2 Fortran 規格に採用された機能

HPF 1.1 の一部であった次の機能は、ISO Fortran の一部となったために本書からは削除された:

- 単純 FORALL 文および構文
- 手続の PURE 属性
- MINLOC と MAXLOC 組込み関数に省略可能な引数 DIM を追加するという拡張

1.4.3 HPF 2.0 で除去または制約された機能

次の機能は言語から削除された:

- 順序的配列は明示的にマップされることはない。
- 分散されたデータの再分散が必要な手続の呼出しでは、手続は明示的引用仕様をもたなければならない。
- INHERIT 指示文の扱いが、INHERIT と DISTRIBUTE の両方を同時に指定することができないように単純化された。
- ポインタの扱いが単純化された。

1.4.4 公認拡張仕様に移された機能

DYNAMIC 属性、REDISTRIBUTE 文、REALIGN 文が公認拡張仕様に移された。

1.4.5 HPF 2.0 の新機能

次の新しい機能が HPF 2.0 に導入された。

- INDEPENDENT ループの REDUCTION 節
- 新 HPF_LIBRARY 手続である SORT_DOWN と SORT_UP

1.4.6 新公認拡張仕様

公認拡張仕様は HPF 1.1 にはない次の機能を含んでいる:

- **部分プロセッサ**への実体のマッピング
- ポインタと構造型成分の明示的マッピング
- 新分散形式: GEN_BLOCK と INDIRECT

- 新指示文: RANGE、 SHADOW、 ON、 RESIDENT、TASK_REGION
- 追加の組込み手続: ACTIVE_NUM_PROCS、ACTIVE_PROCS_SHAPE、および汎用 TRANSPOSE 組込み関数
- 新 HPF_LIBRARY 手続: HPF_MAP_ARRAY と HPF_NUMBER_MAPPED。 HPF_ALIGNMENT、HPF_DISTRIBUTION および HPF_TEMPLATE の改訂
- 新しい WAIT 文および Fortran の READ/WRITE 文への入出力制御パラメタ追加による非同期入出力のサポート
- C および FORTRAN77 との相互引用性をサポートするための EXTRINSIC 機能の拡張

1.4.7 外部機関で保持される認知された HPF 外来仕様

本書では、二つの外部機関で保持される HPF 外来インタフェースが認知されている。

- HPF_CRAFT: SPMD パラダイムを HPF 機能に装備する。
- Fortran 77 ローカルライブラリ: Fortran 77 手続をローカルモードで呼び出すライブラリを定義している。

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 指示文の構文について記述する。

2.1　記述法

本書では 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 Fortran 規格と同じ様式で与えられる。HPF 構文規則と Fortran 構文規則を区別するために、各 HPF 規則は Hsnn 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 Fortran 規格の中で定義されている。また、「記憶単位」のようないくつかの技術用語は Fortran 規格で定義されていることにも注意されたい。

第 1 章で記したように、文中の F95:2.4.7 の形式で示される参照は、Fortran 95 規格の 2.4.7 への参照を表している。

第 III 部は公認拡張仕様を記述する。ここでは以前の章ですでに導入されている構文規則を拡張しなければならない場合がある。特に、ここでの構文規則はしばしば第 II 部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) -*extended* がついている。したがって、*name* 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* のすべての引用は *name-extended* で置き換えられるという条件の下で、それは *name-extended* として引用される。

第 II 部の制約などが公認拡張仕様によって修正される場合には、文章中にそのように記述され、前方への参照が与えられる。そのような前方参照が目立つように、余白に (ここに示すように) 下向き矢印が表示される。 ⇓

(第 III 部の) 各修正は、修正される第 II 部の元の言語定義への後方参照を含む。このような後方参照が目立つように、余白に (ここに示すように) 上向き矢印が表示される。 ⇑

> 【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 指示文は次の意味で Fortran 構文と一貫性がある。すなわち、ある HPF 指示文が将来の Fortran 規格の一部として採用された場合、HPF プログラムを変換するために必要な変更は指示文先頭語 (directive-origin) を空白に置き換えるだけである。

H201 *hpf-directive-line* **is** *directive-origin hpf-directive*

H202 *directive-origin* **is** `!HPF$`
or `CHPF$`
or `*HPF$`

H203 *hpf-directive* **is** *specification-directive*
or *executable-directive*

H204 *specification-directive* **is** *processors-directive*
or *align-directive*
or *distribute-directive*
or *inherit-directive*
or *template-directive*
or *combined-directive*
or *sequence-directive*

H205 *executable-directive* **is** *independent-directive*

制約: *hpf-directive-line* には、他の文があってはならない。

制約: *specification-directive* は、宣言構文 (*declaration-construct*) が現れてもよい場所にのみ現れてもよい。

制約: *executable-directive* は、実行構文 (*executable-construct*) が現れてもよい場所にのみ現れてもよい。

制約: *hpf-directive-line* は、プログラム単位内の周囲の形式に従って、Fortran の自由形式 (F95:3.3.1.1) または固定形式 (F95:3.3.2.1) の注釈行の規則のどちらかに従う。(F95:3.3)

hpf-directive は、文字種 (英大文字、英小文字) を区別せず、自由形式の空白の規則 (3.3.1) に従う。これは、固定形式の HPF プログラムの中でもそうである。しかし、HPF 規格合致の言語処理系は、HPF 指示文の中にある余分なまたは足りない空白を見つけ出す必要はない。Fortran の規則によって、自由形式の *directive-origin* は `!HPF$` でなければならないことに注意されたい。HPF 指示文は継続してよい。その場合、各継続行もまた *directive-origin* で始まらなければならない。継続する HPF 指示行の間に他の文を置いてはならない。HPF 指示行は継続行に現れてはならない。HPF 指示行は後に注釈があってもよい。

どちらのプログラム形式でも、キーワード END FORALL および NO SEQUENCE の中にある空白は任意である。

自由形式の HPF 指示文の継続の例:

```
!HPF$ ALIGN ANTIDISESTABLISHMENTARIANISM(I,J,K) &
!HPF$         WITH ORNITHORHYNCHUS_ANATINUS(J,K,I)
```

以下に、固定形式の HPF 指示文の継続の例を示す。継続を意味する場合以外、第 6 けたは空白でなければならないことに注意する。

```
!HPF$ ALIGN ANTIDISESTABLISHMENTARIANISM(I,J,K)
!HPF$*WITH ORNITHORHYNCHUS_ANATINUS(J,K,I)
```

次の例は、自由形式としても固定形式としても扱える「汎用的」な HPF 指示文の継続方法を示している。最初の行の “&” は第 73 けたにあることに注意されたい。

```
!HPF$ ALIGN ANTIDISESTABLISHMENTARIANISM(I,J,K)                          &
!HPF$&WITH ORNITHORHYNCHUS_ANATINUS(J,K,I)
```

第 III 部では、HPF 2.0 の公認拡張仕様として、新しい宣言指示文と実行指示文を導入する。以下にこれらを記す。

H206 *specification-directive-extended* **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 *range-directive*
or *shadow-directive*

H207 *executable-directive-extended* **is** *independent-directive*
or *realign-directive*
or *redistribute-directive*
or *on-directive*
or *resident-directive*

以下の規則は、Fortran 95 の規則 R215 の拡張である。

H208 *executable-construct-extended* **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*

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。

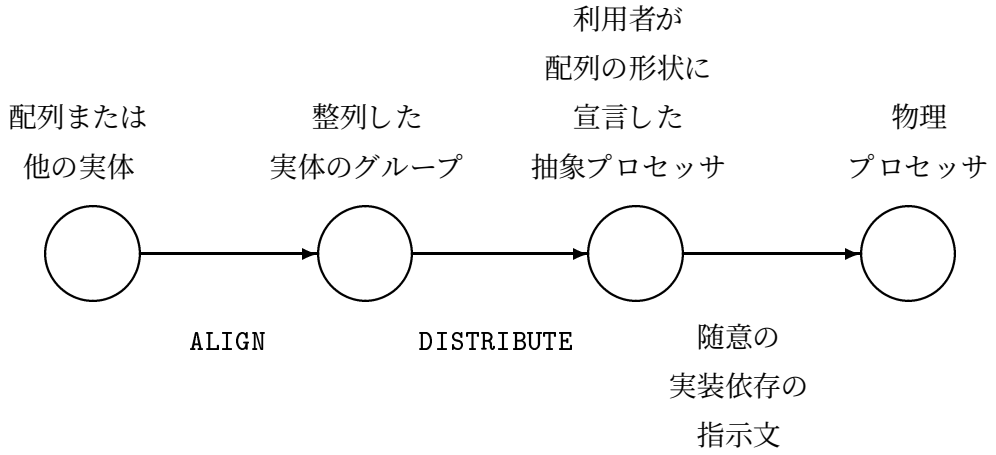
第3章　データマッピング

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】

第4章　副プログラム境界でのデータマッピング

この章では、「呼出し元は...しなければならない」という言い回しは、実装 (すなわち、コンパイラにより生成されるコード) に対する制約を表すもので、プログラマの記述するソースコードに対するものでない。

4.1　はじめに

この節では、マッピング指示文が、副プログラムへ受け渡される引数とどのように関連するのかについての概要を示す。ただし、ここで用いられる言葉は、完全なものではない。厳密な規則はこの節に続くいくつかの節において説明する。

第 3 章で述べたデータマッピング機能に加えて、HPF では仮引数のマッピングに関して多くのオプションが用意されている。

各々の仮引数のマッピングは、それを呼び出す主プログラム又は手続 (“呼出し元”) での対応する実引数と、いくつかの異なった方法で関連づけられる。このために、仮引数に対するマッピング指示文は、3 通りの異なる構文的な形をとることができる。すなわちそれは、**指令的**、**記述的**、**転写的**の 3 つである。

HPF の提供するこれら 3 つの記法は、プログラマが、データをそのままの場所に置いておくのか、あるいは呼出しの間だけ、新しい、そしておそらくは呼ばれた側の副プログラムの実行がより効率よく行なえるようなマッピングに、データを自動的に再マップするのかを指定できるようにする。

これらの記法の意味は次の通りである。

指令的 指令的な指示文は、仮引数のマッピングについて指定する。ただし、実引数はそれと同じマッピングを持たなくても構わない。同じマッピングを**持たない**場合、指定された通りに引数を再マップするコードを生成し、副プログラムの出口で元のマッピングに戻すコードを生成することは、コンパイラの責任となる。このコードは、呼出し元に生成される場合もあれば、呼ばれた側の副プログラム中に生成されることもあるだろう。第 4.6 節で説明する明示的引用仕様の必要性により、マッピングをどちらの側で行なうにしても、必要な情報はコンパイル時に得ることが可能である。

指令的な指示文は、構文的にはプログラムの他の部分に現われる指示文と同じである。例えば、A が仮引数だとすると、

```
!HPF$ DISTRIBUTE A (BLOCK, CYCLIC)
```

は、指令的な指示文となる。

記述的 記述的な指示文は、指令的な構文と全く同じ意味を持つ。違うのは、記述的な指示文は、実引数が再マッピングを必要としないというプログラマによる弱い表明になるということである。

この表明が、“弱い” と表されるのは、もしその表明が偽であっても、プログラムは依然として標準規格合致であるからである。そのような場合、コンパイラは、適切な再マッピングを生成しなければならない。

もしコンパイラが、その表明が偽だと判定できた時、あるいはその表明が真であるかどうか確定できない時は、警告か、診断情報メッセージを出力できる。

> 【利用者への助言】 指令的な指示文に対して記述的な指示文があることの目的は、プログラムの開発及びデバッグのために有用な情報を、コンパイラがプログラマに報告する手段を、単に提供しているにすぎない。記述的な指示文の使用に対していかなる診断メッセージを出すことも、この言語の移植性のある機能ではないことに注意されたい。特に、再マッピングの必要性がないにもかかわらず、コンパイラがこのこと実を確かめるのが不可能か又は自明でない場合があることに注意が必要である。コンパイラが異なれば、違った状況でメッセージを出力する場合もある。さらに、そのようなメッセージを出力しなければならないという要求も全くない。【以上】

記述的な指示文は、マッピングの記述の前にアスタリスクが付くことを除いては、指令的な指示文と同じ形をしている。

```
!HPF$ DISTRIBUTE A *(BLOCK, CYCLIC)
```

は、記述的な指示文である。

転写的 マッピングは、指定されない。呼ばれた側の副プログラムは、渡ってきた引数がどのようにマップされていてもそれを受け入れる必要がある。もちろんこのことは、呼出し元 (の実装) がマッピングに関する情報を実行時に渡すことを意味する。

転写的な指示文は、分散形式及びプロセッサ構成を指定するところに単にアスタリスクを書くことで記述できる。

```
!HPF$ DISTRIBUTE A *
!HPF$ DISTRIBUTE B * ONTO *
```

は、転写的な指示文である。転写的な整列を指定するのには、INHERIT 指示文 (第 4.4.2 項を参照) が用いられる。

分散形式と、プロセッサ構成の両方に、指令的、記述的、複写的の指示ができる。整列は、より複雑である。なぜなら、仮引数が整列すべきテンプレートが必要だからである。このテンプレートは、指定しなくても構わない (この場合もちろん ALIGN 指示文は存在しない)。この時、仮引数のテンプレートは、**自然テンプレート** になる。(“自然テンプレート” は、この後の第 4.4.1 項で定義される。) そうでない場合は、次のそれぞれの項目の内の一つが成り立たなければならない。

- テンプレートは、指令的な ALIGN 指示文により明示的に指定される
- テンプレートは、記述的な ALIGN 指示文により明示的に指定される
- テンプレートは、継承される。これは、仮引数に INHERIT 属性 (この後の第 4.4.2 項で説明されている) を与えることで指定される。これは、テンプレートが、対応する実引数が最終的に整列されているテンプレートのコピーとなることを暗黙に指定する。さらに、仮引数のそのテンプレートへの整列は、対応する実引数のそれと同じになる。これは結局、整列の転写的な形式と同じ効果を持つ。

これについては、この後の第 4.4.1 項でより正確に説明されている。

【利用者への助言】 例えば、一部分が指令的で、一部分が転写的であるような混合したマッピング指示文を記述することは可能だが、そうすることにはおそらく何のメリットもないだろう。これらの指示文のポイントは、コンパイラが、必要な再マッピングをすべて正しく、効率的に処理できるようにするところにある。再マッピングは、次のような理由が原因で起こることがある。

- 実引数と仮引数の整列状態を合わせるため
- 実引数と仮引数の分散状態を合わせるため
- 実引数と仮引数が分散されるプロセッサ構成を一致させるため

ほとんどのマシンにとって、これらのどの理由による再マッピングの間にも、実質的なコストの差はない。それゆえ、マッピング指示を純粋に転写的な指示にする、純粋に指令的な指示にする、あるいは純粋に記述的な指示にするほうが、よい選択といえるであろう (少なくともそのほうが読みやすくなる)。

転写的な指示によるマッピングは、ライブラリを書く場合に有用かもしれないが、副プログラム中でランタイムコストを発生することになる。それゆえ、通常のユーザーコード中でこれらを使用することは避けるべきである。【以上】

4.2 どのような再マッピングが必要とされ、誰がそれを行なうのか

呼び出される副プログラムに対する明示的引用仕様が宣言されており、その引用仕様が仮引数に対する指令的又は記述的なマッピング指示文を含み、かつ対応する実引数の再マッピングが必要な場合、その副プログラムの呼出しは、データが、明示された引用仕様で指示された仮引数のマッピングにマッチするようなテンポラリの変数に、あたかもコピーされたかのようにして行なわれる。仮引数のテンプレートは、引用仕様に記述された通りになる。

明示的引用仕様が存在しない場合は、再マッピングは必要ない。これは、第 4.6 節で示される要求の結果である。

簡単のため、この章で示されるコード片には、そのように必要とされる明示的引用仕様が全て含まれているわけではないことに注意されたい。

基本となる原理は、**引数の再マッピングは、呼出し元からは見えない**ということである。つまり、副プログラムからの戻り時や、呼出し元の実行が再開された時には、その呼出し後に呼出し元がアクセスできる全ての実体は、呼出し前と全く同様にマップされているという

ことである。手続は、それを呼び出した側からも見えるようないかなる実体のマッピングを変更することもできない。

⇓ 【利用者への助言】いくつかの公認拡張では、この制約が緩和される。例えば、第 8.6 節や第 8.8 節を参照してほしい。【以上】

4.3 分散とプロセッサ構成

全ての *distributee* が仮引数であるような DISTRIBUTE 指示文では、*dist-format-clause* か *dist-target* あるいはその両方を、アスタリスクで始めたり、アスタリスクだけから構成したりできる。

- アスタリスクがない場合、*dist-format-clause* 又は *dist-target* は、指令的になる。この節は分散を記述し、言語処理系にその分散に従うよう要求する。これによって、要求された仮引数の分散を実現するために、実行時に実引数の再マッピング又はコピーが、呼出し元か呼ばれた側の副プログラム (の実装) で行なわれる可能性がある。
- アスタリスクで始めると、*dist-format-clause* 又は *dist-target* は、記述的になる。この指示文は、次の点を除いて、指令的な指示文と全ての点において同等である。すなわち、コンパイラが、実引数の再マッピングが必要ないということが判定できなかった場合、そのことについての診断メッセージを出力できるということである。この点についての詳しい説明は、第 4.1 節を参照してほしい。
- アスタリスクのみからなると、*dist-format-clause* 又は *dist-target* は、転写的になる。この節は、分散については何も述べないが、言語処理系に対して、実引数の分散をコピーすることを要求する。(この意図は、引数が参照渡しの場合、実行時にデータの移動が不要になるということである。)

一つの DISTRIBUTE 指示文の中で、*dist-format-clause* がアスタリスクを持ち、*dist-target* が持たなくてもよく、またその逆も可能である。

4.3.1 例

以下の仮引数用の DISTRIBUTE 指示文の例は、いろいろな組合せについて説明している。

```
!HPF$ DISTRIBUTE URANIA (CYCLIC) ONTO GALILEO
```

言語処理系は、プロセッサ構成 GALILEO へ URANIA を CYCLIC 分散するために必要なことをしなければならない。

```
!HPF$ DISTRIBUTE POLYHYMNIA * ONTO ELVIS
```

言語処理系は、POLYHYMNIA を、その現在の分散形式 (それが何であれ) に従って、プロセッサ構成 ELVIS へ分散するために必要なことをしなければならない。(POLYHYMNIA は他のプロセッサ構成の可能性もある。)

```
!HPF$ DISTRIBUTE THALIA *(CYCLIC) ONTO *FLIP
```

言語処理系は、プロセッサ構成 FLIP へ THALIA を CYCLIC に分散するために必要なことをしなければならない。プログラマは、実引数がすでにそのように分散されており、再マッピングは必要ないことを知っている。

```
!HPF$ DISTRIBUTE EUTERPE (CYCLIC) ONTO *
```

言語処理系は、実引数が分散されていたプロセッサ構成上に EUTERPE を CYCLIC 分散するために、必要なことをしなければならない。

```
!HPF$ DISTRIBUTE ERATO * ONTO *
```

ERATO のマッピングは、実引数のマッピングから変更してはいけない。

DISTRIBUTE ERATO * ONTO *は、

```
!HPF$ DISTRIBUTE ERATO (*) ONTO *
```

とは同じ意味を持たないことに注意されたい。後者は、ERATO を実引数が分散されていたプロセッサ構成上に、*(つまり、オンプロセッサ) 分散するという意味になる。この場合は、プロセッサ構成は必然的にスカラになる。

4.3.2 省略時の解釈

仮引数については、*dist-format-clause* 又は *dist-onto-clause* のどちらか一方を省略しても構わない。これは、次のように解釈される。

仮引数が INHERIT 属性 (第 4.4.2 項参照) を持っている時は、いかなる場合においても分散指示をすることは許されない。分散も、整列と同じく実引数から引き継がれる。

その他のケースで、分散情報が省略されている場合、コンパイラは任意に分散形式又はターゲットとなるプロセッサ構成を選択できる。

以下に 2 つの例を示す。

```
!HPF$ DISTRIBUTE WHEEL_OF_FORTUNE *(CYCLIC)
```

プログラマは、WHEEL_OF_FORTUNE 仮引数に対応する実引数がすでに CYCLIC 分散されていることを知っている。コンパイラは、渡ってきたデータが本当に CYCLIC かどうか確かめ、もしそうでない場合は、必要ならば再マッピングを行なわなければならない。データは、何か他のプロセッサ構成に再マップすることもできるが、そうする理由はない。もしそのような再マッピングを行った場合は、プログラマを驚かせることになるであろう。

```
!HPF$ DISTRIBUTE ONTO *TV :: DAVID_LETTERMAN
```

プログラマは、DAVID_LETTERMAN 実引数に対応する仮引数が、TV に、何らかの方法で分散されていることを知っている。コンパイラは、それが本当かどうか確かめ、本当でないなら、そうなるようにする必要がある。分散形式は、DAVID_LETTERMAN が TV 上に保持される限り、変更できる。(この宣言は、属性形式で記述されなければならないことに注意されたい。

```
!HPF$ DISTRIBUTE DAVID_LETTERMAN ONTO *TV          ! 規格合致でない
```

という文形式は、DISTRIBUTE 指示文の構文に適合しない。)

4.4 整列

4.4.1 仮引数のテンプレート

ここで、仮引数が最終的に整列されているテンプレートがどのように決められるのかについて詳細に述べる。

テンプレートは、副プログラムの引数引用仕様を通して渡されることはない。仮引数と、それに対応する実引数が同じテンプレートに整列するのは、テンプレートが、親子結合や参照結合により、呼出し元と呼ばれた副プログラム側の両方からアクセス可能な場合のみである。その他の場合は、仮引数が整列するテンプレートと実引数が整列するテンプレートは常に異なる。ただし、テンプレートがコピーされたものである場合はあり得る (第 4.4.2 項参照)。HPF の実装では、手続からの戻りの際に、実引数が、手続呼出しの前に整列していたテンプレートと同じテンプレートに整列される。

仮引数のテンプレートは、次の三つのいずれかの方法で作成される。

- 仮引数が `ALIGN` 指示文の中で *alignee* として明示的に使用されている場合、*align-target* がテンプレートであれば、仮引数のテンプレートはその *align-target* である。そうでなければ、仮引数のテンプレートは、*align-target* が最終的に整列されているテンプレートである。

- 仮引数が明示的に整列されておらず、しかも `INHERIT` 属性 (第 4.4.2 項で説明されている) を持っていない場合、そのテンプレートは、仮引数と同じ形状と上下限を持ち、その仮引数の自然テンプレートと呼ばれる。

  (よって、第 4.3 節の全ての例では、自然テンプレートが使用されていることになる。)

- 仮引数が明示的に整列されておらず、しかも `INHERIT` 属性を持っている場合、そのテンプレートは次の規則にしたがって実引数から「継承」される。

  – 実引数が全体配列である場合、仮引数のテンプレートは、実引数が最終的に整列されているテンプレートのコピーである。
  – 実引数がどの添字もベクトル添字でないような配列 A の部分配列である場合、仮引数のテンプレートは、A が最終的に整列するテンプレートのコピーである。
  – 実引数がその他の式である場合、テンプレートの形状と分散は、言語処理系が自由に選択できる（したがって、プログラマはその分散について前もって何も知ることはできない)。

  これらすべての場合において、仮引数は継承 (`inherited`) テンプレートを持つと言う。

4.4.2 INHERIT 指示文

`INHERIT` 指示文は、仮引数を、対応する実引数のテンプレートのコピーへ、実引数が整列されているのと同じ方法で整列することを指定する。

H401 *inherit-directive* **is** `INHERIT` *inheritee-list*

H402 *inheritee* **is** *object-name*

制約: *inheritee* は仮引数でなければならない。

制約: *inheritee* は *alignee* であってはならない。

制約: *inheritee* は *distributee* であってはならない。

> 【利用者への助言】 公認拡張では、上記の 3 つの制約の内の一つ目は、ポイントに対しては緩和される (第 8.8 節を参照してほしい)。【以上】 ⇓

INHERIT 指示文は、指定した副プログラムの仮引数に INHERIT 属性を持たせる。INHERIT 属性を持てるのは仮引数だけである。ある実体が INHERIT 属性と ALIGN 属性の両方を持つことはできない。ある実体が INHERIT 属性と DISTRIBUTE 属性の両方を持つことはできない。INHERIT 指示文は、有効域の *specification-part* の中でだけ使用できる。

INHERIT 属性は、仮引数のテンプレートが、実引数のテンプレートのコピーを作成することによって継承されたものでなければならないことを指定する。さらに、INHERIT 属性を指定された引数は、他の明示的なマッピング指示文を指定することはできない。INHERIT 属性は、継承テンプレートに対する DISTRIBUTE * ONTO * の分散を意味する。ゆえに、最終的な効果は、コンパイラに、データを正確にそのままの位置にとどめ、実引数を再マップしてはならないということを伝えることにある。仮引数は、実引数と全く同じ方法でマップされる。副プログラムは、実引数が論理プロセッサにどのようにマップされていても、正しく動くようにコンパイルされなければならない。

A が、配列仮引数である場合、

```
!HPF$ INHERIT A
```

という指示文は、

```
!HPF$ DISTRIBUTE A * ONTO *
```

よりもより一般的であることに注意されたい。その理由は次の通りである。INHERIT 指示文は、A が整列している継承テンプレートが、* ONTO*で分散されるが、A はそのテンプレートに、何らかの自明でない方法で整列するということを意味する。一方、DISTRIBUTE 指示文は、A は明白にその自然テンプレートに整列し、その後* ONTO*で分散されるという意味になる。

例えば、次のようなコードは許されない。

```
!HPF$ PROCESSORS P(2)
      REAL, DIMENSION(100) :: A
!HPF$ DISTRIBUTE (BLOCK) ONTO P :: A

      CALL FOO(A(1:50))


      ...

      SUBROUTINE FOO(D)
      REAL, DIMENSION(50) :: D
!HPF$ DISTRIBUTE D *            ! 規格合致でない
```

D に対する転写的な指示文は、D の自然テンプレートが BLOCK 分散されていないため、規格合致していない。一方、この間違った指示文を次の指示文で置き換えることは、正当な処置である。

```
!HPF$ INHERIT D
```

4.4.2.1 例

ここで、INHERIT の使用法について簡単な例をあげる。

```
      REAL DOUGH(100)
!HPF$ DISTRIBUTE DOUGH(BLOCK(10))
      CALL PROBATE( DOUGH(7:23:2) )
      ...
      SUBROUTINE PROBATE(BREAD)
      REAL BREAD(9)
!HPF$ INHERIT BREAD
```

BREAD の継承テンプレートは形状 [100] を持つ。要素 BREAD(I) は継承テンプレートの要素 5 + 2*I に整列し、そのテンプレートは BLOCK(10) の分散を持つ。

もっと複雑な例も簡単に作ることができる。継承テンプレートは、仮引数の次元数とは異なった次元数を持つことができ、さらにそれは実引数の次元数とも異なった次元数を持ってもよいということを気にとめておくことは重要である。例えば、次のような部分がプログラムに含まれていたとする。

```
      REAL A(100,100)
!HPF$ TEMPLATE T(100,100,100)
!HPF$ DISTRIBUTE T(BLOCK,CYCLIC,*)
!HPF$ ALIGN A(I,J) with T(J,3,I)
      CALL SUBR(A(:,7))
      ...
      SUBROUTINE SUBR(D)
      REAL D(100)
!HPF$ INHERIT D
```

この場合、仮引数 D は 1 次元である。D は、2 次元の実引数 A の 1 次元部分に対応し、A は、3 次元テンプレート T の 2 次元断面に整列する。D のテンプレートは、この 3 次元テンプレートのコピーである。D は、この継承テンプレートの (7, 3, :) の部分に整列する。よって、仮引数 D の「見える」次元は、*に分散される。しかし、例えばもし CALL 文が、

```
      CALL SUBR(A(7,:))
```

の形だった場合、仮引数の「見える」次元は、BLOCK に分散される。

4.4.3 記述的な ALIGN 指示文

align-spec の先頭にアスタリスクがあるかないかは、*dist-format-clause* の場合と同じ意味を持つ。つまり、アスタリスクがあるかないかは、ALIGN 指示文が記述的なのか指令的なのかをそれぞれ指定する。

*で始まらない *align-spec* が仮引数に適用された場合、その意味は、その仮引数が副プログラムへの入口で強制的に指定の整列を持たされるということである。この場合、呼出し元かもしくは副プログラム側 (の実装) で、実引数のデータ又はそのコピーの一時的な再マッピングを必要とする場合がある。

仮引数は、*align-target* としても使用できることに注意されたい。

```
      SUBROUTINE NICHOLAS(TSAR,CZAR)
      REAL, DIMENSION(1918) :: TSAR,CZAR
!HPF$ INHERIT :: TSAR
!HPF$ ALIGN WITH TSAR :: CZAR
```

この例では、最初の仮引数 TSAR は対応する実引数に整列したままとなるが、2 番目の仮引数の CZAR は強制的に最初の仮引数へ整列させられる。二つの実引数がすでに整列していれば、実行時にデータの再マッピングは必要ない。そうでなければ、何らかの再マッピングが起きるであろう。

align-spec が “*” で始まる場合、*alignee* は仮引数でなければならない。“*” は、実引数がすでに指定された整列をしており、実行時にそれを再マップする必要はないということを、プログラマが知っているということを示している。(前に述べた通り、このプログラマの知識が正しくなければならないという要求はない。コンパイラは、指令的な整列の場合と同じく、必要と思われるならば、再マッピングを生成しなければならない。) 例えば、上の例で、ALIGN 指示が次のように変更されたとする。

```
!HPF$ ALIGN WITH *TSAR :: CZAR
```

これは、プログラマが、TSAR に対応する実引数の再マッピングは必要ないということを表明していることになる。

単に “ALIGN WITH *” という言い方は許されない。アスタリスクの後に *align-target* がなければならない。(「どの整列でも受け入れる」と言う正しい方法は、INHERIT である。)

仮引数が明示的な ALIGN 属性又は DISTRIBUTE 属性を持たない場合、コンパイラは暗黙の整列と分散を指定する。それは、「表明のアスタリスク」なしに明示的に記述できるものの中から選択される。

4.4.3.1 例

INHERIT を使用しない場合、副プログラム境界で再マッピングが起きないことを保証するには、仮引数の明示的な整列が必要となるであろう。次の例を考えてもらいたい。

```
      LOGICAL FRUG(128)
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ DISTRIBUTE (BLOCK) ONTO DANCE_FLOOR::FRUG
      CALL TERPSICHORE(FRUG(1:40:3))
```

部分配列 FRUG(1:40:3) は、次のような方法で論理プロセッサへマップされる。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1			25												
	10			34											
		19													
4			28												
	13			37											
		22													
7			31												
	16			40											

まず、サブルーチン TERPSICHORE の引用仕様が次のようだったとする。

```
      SUBROUTINE TERPSICHORE(FOXTROT)
      LOGICAL FOXTROT(:)
!HPF$ INHERIT FOXTROT
```

FOXTROT のテンプレートは、全体配列 FRUG の 128 要素からなるテンプレートのコピーである。このテンプレートは次のようにマップされる。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	9	17	25	33	41	49	57	65	73	81	89	97	105	113	121
2	10	18	26	34	42	50	58	66	74	82	90	98	106	114	122
3	11	19	27	35	43	51	59	67	75	83	91	99	107	115	123
4	12	20	28	36	44	52	60	68	76	84	92	100	108	116	124
5	13	21	29	37	45	53	61	69	77	85	93	101	109	117	125
6	14	22	30	38	46	54	62	70	78	86	94	102	110	118	126
7	15	23	31	39	47	55	63	71	79	87	95	103	111	119	127
8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128

FOXTROT(I) は、このテンプレートの要素 3*I-2 に整列する。

一方、TERPSICHORE の引用仕様が、代わりに次のようだったと仮定する。

```
      SUBROUTINE TERPSICHORE(FOXTROT)
      LOGICAL FOXTROT(:)
!HPF$ DISTRIBUTE FOXTROT(BLOCK)
```

この場合、FOXTROT のテンプレートは、FOXTROT 自身と同じ大きさ 14 の自然テンプレートである。実引数 FRUG(1:40:3) は、次のような方法で 16 のプロセッサにマップされる。

Abstract processor	Elements of FRUG
1	1, 2, 3
2	4, 5, 6
3	7, 8
4	9, 10, 11
5	12, 13, 14
6–16	none

すなわち、仮引数の要素の (実引数のテンプレート上での) 位置は、次のようになる。

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1			9												
	4			12											
		7													
2			10												
	5			13											
		8													
3			11												
	6			14											

このようなレイアウト (最初のプロセッサに 3 要素、2 番目に 3 要素、3 番目に 2 要素、4 番目に 3 要素、...) は、BLOCK 分散として適切に表すことはできない。それゆえ、呼出しの際に再マッピングが起きるであろう。

仮引数を、大きさ 128 の BLOCK 分散されたテンプレートを宣言し、それに明示的に整列すれば、INHERIT を使わずに、再びマップを防ぐことができる。

```
      SUBROUTINE TERPSICHORE(FOXTROT)
      LOGICAL FOXTROT(:)
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK) ONTO DANCE_FLOOR::GURF(128)
!HPF$ ALIGN FOXTROT(I) WITH GURF(3*I-2)
```

【利用者への助言】後者のテクニックの利点は、それが使用できるところであれば、コンパイラにより多くの情報を与えられるという点である。この情報は、より効率的なコードを生成するのに使用することができる。【以上】

4.5 マッピングの集合における同値関係と半順序

名前付き実体のマッピングの集合は、ある種の同値関係を法とする半順序を持つ。おおまかにいうと、P と Q が二つのマッピングである時、Q が P の「特殊化」であるということ (すなわち、この順序関係において「Q が P の下にある」ということ) は、P が部分的に指定さ

れており、Q は、P を満たすようなマッピングの一つであるということである。この記法は、この後の第 4.6 節と　第 8.8 節において用いられる。

【利用者への助言】 これらの条件に対する説明は非常に複雑なので、もしあなたが常に明示的引用仕様を指定し (この後で説明されるように、これは非常に簡単で、たいていの場合自動的にできる)、マッピングされたポインタ (第 8.8 節で説明されている承認された拡張である) を使わないならば、この章を読み飛ばして構わない。【以上】

⇓

詳細な定義を以下に示す。
まず、*dist-format* 指定子の同値性について定義する。

1. 「同値である」という表現に記法 $\equiv$ を用いると、

$$
\begin{aligned}
\texttt{BLOCK} &\equiv \texttt{BLOCK} \\
\texttt{CYCLIC} &\equiv \texttt{CYCLIC} \\
* &\equiv * \\
\texttt{BLOCK}(n) &\equiv \texttt{BLOCK}(m) \quad ! \, m \text{ と } n \text{ の値が等しいとき、かつそのときに限り} \\
\texttt{CYCLIC}(n) &\equiv \texttt{CYCLIC}(m) \quad ! \, m \text{ と } n \text{ の値が等しいとき、かつそのときに限り} \\
\texttt{CYCLIC} &\equiv \texttt{CYCLIC(1)}
\end{aligned}
$$

2. これ以外の、字面上異なる *dist-format* 指定は、同値ではない。

これは、一般の数学的な意味での同値関係と同じである。

次に、マッピングの半順序を定義する。S (“special”) と G (“general”) を二つのデータ実体とする。

次のいずれかの条件が成り立つ場合、かつその場合に限り、S のマッピングは、G のマッピングの**特殊化**であるという。

1. G は INHERIT 属性を持つ
2. S は INHERIT 属性を持たず、次の全ての条件が成り立つ
   (a) S は名前付き実体である
   (b) S と G の最終整列先が同じ形状を持つ。かつ、
   (c) S と G の対応する次元が、それぞれの最終整列先の対応する次元にマッピングされ、S と G の対応する要素が、それぞれの最終整列先の対応する同じ要素に整列する。かつ、
   (d) 次のいずれかが成り立つ
      i. S と G の最終整列先が明示的に分散されていない。または、
      ii. S と G の最終整列先が両方とも明示的に分散されている。この場合、G の最終整列先に対する分散指示文は、次の条件の内の一つを満たさなければならない。
         A. *dist-onto-clause* を持たない。または、
         B. *dist-onto-clause* が “ONTO *” である。または、
         C. S の最終整列先に対して明示的に指定された分散指示文で指定されたのと同じ形状を持つプロセッサ構成を指定する *dist-onto-clause* を持つ

      さらに次の条件の内の一つを満たさなければならない。
         A. *dist-format-clause* を持たない。または、

B. *dist-format-clause* が “*” である。または、
C. 各 *dist-format* は、S の最終整列先に対する明示的分散指示文の *dist-format-clause* の対応する位置の *dist-format* と (上で定義された意味で) 同値である。

この定義によれば、以下のことがいえる。

- 名前付き実体に対する任意のマッピングは、それ自身の特殊化になっている
- A、B、C が名前付き実体で、A が B のマッピングの特殊化になっており、B が C のマッピングの特殊化になっている場合、A は C のマッピングの特殊化になっている

すなわち、名前付き実体に適用される特殊化の関係は反射的かつ推移的であり、それゆえその関係は、名前付き実体のマッピング集合に対する同値関係を作るのに使用できる。つまり、お互いに特殊化の関係にある二つのマッピングは、同値関係にあるということができる。この定義によれば、特殊化の関係が、名前付き実体のマッピング集合に対する、同値関係を法とする半順序を生じさせる。INHERIT マッピングは、この半順序の中の唯一の極大元である。

4.6 明示的引用仕様を省略するための条件

ある条件では、副プログラムの明示的引用仕様は必要ない。マッピング指示を使用した HPF プログラムでは、Fortran において許容された引用仕様の省略の条件はかなり厳しくなる。

【利用者への助言】 これらの条件は複雑である。明示的引用仕様を使用していれば、本節を読む必要はないことを是非承知しておいてほしい。もし何等かの疑問が浮かんだならば、すぐに明示的引用仕様を使用しているか確認してもらいたい。【以上】

以下の全ての条件が満たされている場合**以外は**、明示的引用仕様が必要である。

1. Fortran 規格によって必要とされていないこと。**かつ**
2. 仮引数は、転写的な分散が指示されてなく、INHERIT 属性を持たないこと。**かつ**
3. それぞれの対応する実引数と仮引数は、
   (a) どちらも暗黙のマップされているか、
   (b) どちらも明示的にマップされており、実引数のマッピングが仮引数のマッピングの特殊化になっている。

   かつ、
4. それぞれの対応する実引数と仮引数は、以下のいずれかであること。
   (a) ともに順序的であるか、
   (b) ともに非順序的である。

【仕様の根拠】 これは、以下のような結果をもたらす。

- 純粋な (すなわち、HPF 指示文がない)Fortran プログラムは、少なくとも全ての変数がデフォルトでは順序的であるようなコンパイル環境においては、引用仕様を追加しなくても HPF 規格合致でありつづける。これは、項目 1、2、3(a)、および 4(a) によって保証される。

- もし再マッピングが必要ならば、呼出し元はその事実を認識できるであろう。従って実装では全ての再マッピングを呼出し元で実行するよう選択することができる。

【以上】

【利用者への助言】 この要請はユーザに明示的引用仕様を常に記述するよう、強く求めている。これは良いことであり、明示的引用仕様によって多くの誤りをコンパイル時に検出でき、頑健なソフトウェアの開発過程を大いにスピードアップすることができる。

明示的引用仕様は 3 通りの方法で規定されうることに注意されたい。

1. モジュール副プログラムが明示的引用仕様を持つ場合。
2. 内部副プログラムが明示的引用仕様を持つ場合。
3. 明示的引用仕様が引用仕様宣言によって規定される場合。

更に、組込み手続は定義によりいつも明示的引用仕様を持つ。

慣用的な Fortran のプログラミング手法では、モジュールの使用法は広範なものになっている。例えば、どの副プログラムもモジュールに含むことができる。この手法では、プログラマが特別に努力することなく、明示的引用仕様が自動的に提供される。引用仕様宣言を記述する必要があるのは非常に希なはずである。【以上】

4.7 手続の特性

Fortran 規格 (F95:12.2) に与えられる仮データ実体や関数結果の特性は、実体の ***HPF* 特性** (*hpf-characreristics*) をも含むように拡張される。これは、以下のように再帰的に定義される。

- プロセッサ構成は、一つの HPF 特性を持つ。その形状である。
- テンプレートは以下の 3 つの HPF 特性を持つ。
  1. その形状。
  2. 明示的に記述されている場合、その分散。
  3. 明示的に記述されている場合、そのテンプレートが分散されているプロセッサ構成の HPF 特性 (すなわち形状)。
- 仮データ実体は、以下の HPF 特性を持つ。
  1. 明示的に指定されている場合、その整列。およびその整列先の全ての HPF 特性。
  2. 明示的に指定されている場合、その分散。および明示的に指定されている場合、その仮データ実体が分散されているプロセッサ構成の HPF 特性 (すなわち形状)。
- 関数の結果は仮データ実体と同じ HPF 特性を持つ。特に、以下の HPF 特性を持つ。
  1. 明示的に指定されている場合、その整列。およびその整列先の全ての HPF 特性。
  2. 明示的に指定されている場合、その分散。および明示的に指定されている場合、その関数結果が分散されているプロセッサ構成の HPF 特性 (すなわち形状)。

【仕様の根拠】 明示的引用仕様が引用仕様宣言によって与えられた場合、Fortran 規格では、当該引用仕様宣言にどのような情報を指定しなければならないかを規定して

いる。その規定は、Fortran の**特性**の概念を使用して行なわれる。例えば、仮データ実体の特性には、その型が含まれる。特性は引用仕様宣言中に指定しなければならない。Fortran 規格 (F95:12.3.2.1) で、

> 引用仕様本体は、手続の特性のすべてを指定する。これらは、手続の定義と一致していなければならない...

と、規定されている。通常、手続の引用仕様宣言は当該手続内の該当する宣言の字面どおりのコピーである。この章では、単に、そのコピーが手続の仮引数に関する全ての明示的なマッピングの指示を含んでいなければならないと述べているだけである。【以上】

4.8 引数の受け渡しと順序結合

手続呼出しにおける実引数について、Fortran は配列要素 (スカラ) と結合する仮引数が配列であることを許している。更に、仮引数の形状を対応する実引数と異なるものにすることができ、その結果、手続呼出しを通して実引数の形状を変更することも許している。Fortran の記憶列の性質が、仮引数の値を定めるために使われてきた。Fortran 77 から継承されたこの機能は、より大きな配列の部分行または列の開始アドレスを手続に渡すために広く使用されてきた。HPF の配列はプロセッサにまたがってマッピングされる可能性があるため、この機能は完全にはサポートされていない。

4.8.1 順序結合規則

1. 配列要素または大きさ引継ぎ配列の名前を、実引数として使用する場合は、結合する仮引数はスカラであるか、順序的配列の指定がなければならない。
   非順序的配列の配列要素特定子は、配列仮引数と結合してはならない。
2. 実引数が配列または部分配列であり、対応する仮引数と形状が異なる場合は、仮引数は順序的であると宣言されており、実配列引数は順序的でなければならない。
3. (スカラまたは配列の) 文字型の実体は、もしそれが第 3.8.1.1 項の定義 4 を満たしているなら、非順序的である。長さ明示の (explicit-length) 文字型仮引数の長さが、実引数の長さと異なる場合、仮引数と実引数はともに順序的でなければならない。
4. 明示的引用仕様がないとき、順序的実引数は非順序的仮引数と結合することはできない。また、非順序的実引数は順序的仮引数と結合することはできない。(この項目は第 4.6 節の一部の繰り返しにすぎない。)

4.8.2 順序結合についての説明

配列仮引数とそれに結合する配列実引数との形状が異なるとき、実引数は式 (expression) であってはならない。配列値を持つ式 (array-valued expression) が順序的であることを宣言する HPF のメカニズムは存在しない。その様な式を実引数として次元数の異なる仮引数と結合するためには、その実引数を、第 3.8.1.1 項の定義 4 に従って順序的であるよう強制された名前つき配列変数に代入する必要がある。

4.8.3 順序結合の例

以下のようなサブルーチンの一部が与えられたとき、

```
SUBROUTINE HOME (X)
DIMENSION X (20,10)
```

規則 1 により、

```
CALL HOME (ET (2,1))
```

という記述は、X が HOME において順序的であると宣言されており、ET が呼出し元の手続内で順序的であるときだけ正しいものである。

同様に、規則 2 及び 4 により、

```
CALL HOME (ET)
```

という記述は、ET および X が、ともに順序的配列であるか、ET および X が同じ形状を持っており、(明示的引用仕様がない場合には) 同じ順序属性を持っているかのどちらかであることを要請する。

規則 3 は、文字型実体に対して特別な考慮を示している。以下のような手続呼出しにまたがった文字型実体の長さの変化は、Fortran の概念では正しいものである。

```
CHARACTER (LEN=44) one_long_word
one_long_word = 'Chargoggagoggmanchaugagoggchaubunagungamaugg'
CALL webster(one_long_word)

SUBROUTINE webster(short_dictionary)
CHARACTER (LEN=4) short_dictionary (11)
    !例えば、short_dictionary(3) は 'agog'となることに注意
```

HPF においては、実引数と仮引数はともに順序的でなければならない。(例の中の Chargoggagoggmanchaugagoggchaubunagungamaugg は、現在は「ウェブスター湖 (Lake Webster)」と呼ばれているマサチューセッツ州の湖にニプマク族がつけた元の名前である)。

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
```

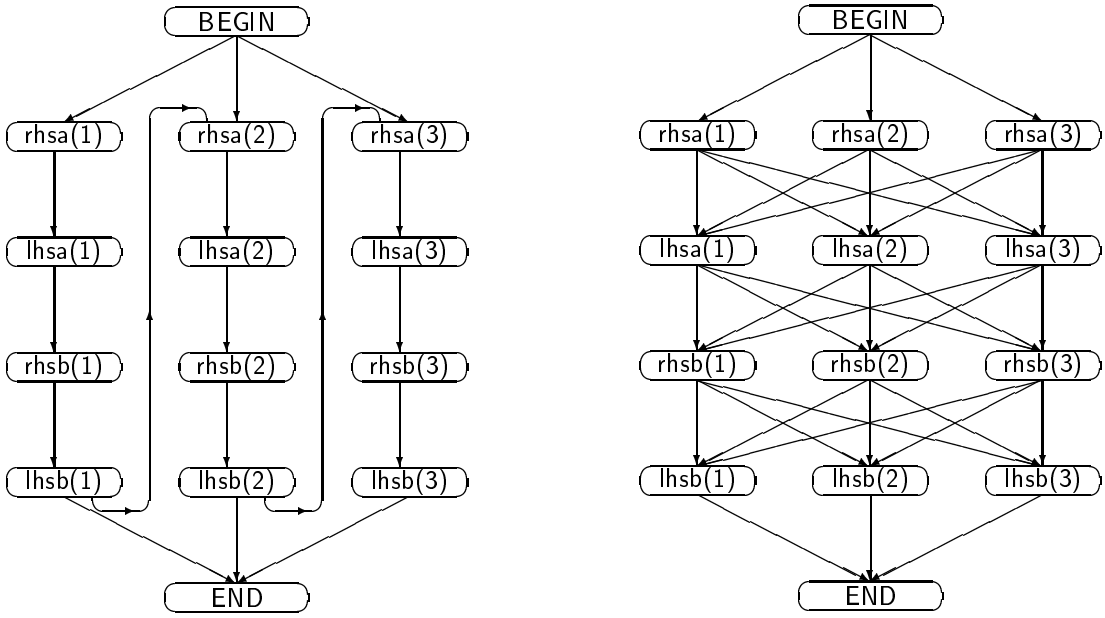
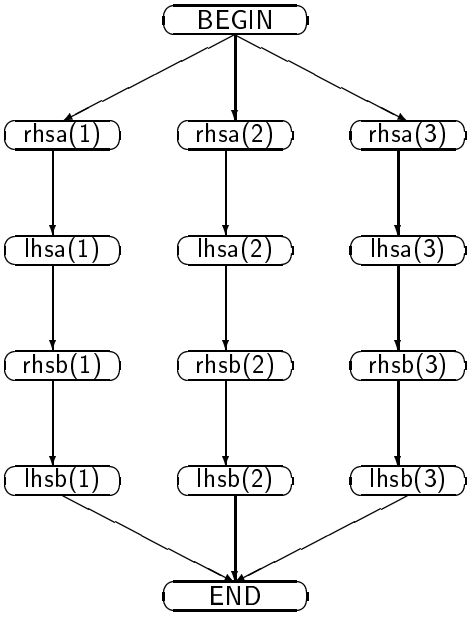


図 5.1: INDEPENDENT 表明を伴わない DO と FORALL の依存関係

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
```

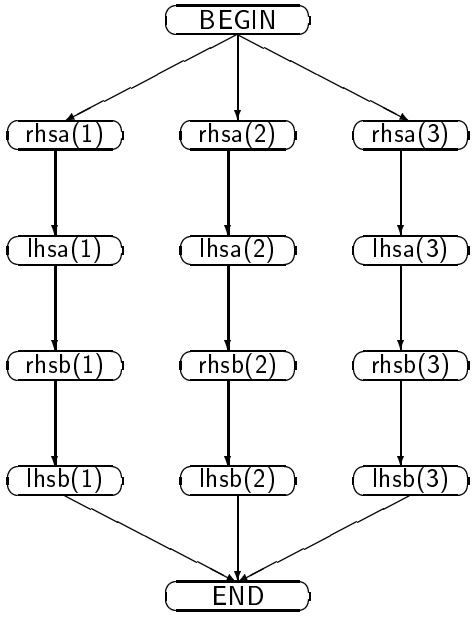


図 5.2: INDEPENDENT 表明を伴う DO と FORALL の依存関係

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
```

第6章　外来プログラム単位

HPF のグローバルな計算モデルは、Fortran を拡張 (および制限) することによって、広範なハードウェアアーキテクチャにおいて効率的に実装することのできる Fortran の計算モデルを、プログラマに提供する。対象とするアーキテクチャは、一般に、複数プロセッサ、非一様アクセス特性を持つ複数メモリ、及び複数の結合網を持つ。この計算モデルは、論理的に単一の制御スレッドを提供しており、配列構文、FORALL 文などの Fortran のデータ並列機能、および、Fortran の有効域規則により定義されるデータ可視性を含んでいる。特にこのモデルは、このようなアーキテクチャを利用するための低レベルの機能、例えばスレッドライブラリや明示的メッセージ通信などを要求しない。プログラマは、HPF コンパイラが、HPF の機能を手助けとして、与えられたハードウェアアーキテクチャに対してデータや計算のマッピングを行い、効率の良いコードを生成すると期待できる。

この章では、HPF で記述されたプログラム単位から、HPF のグローバルモデルを使用しない非 HPF プログラム単位を使用するための、外来 (*extrinsic*) 機構を定義する。非 HPF 手続に対する明示的引用仕様をどのように記述するかを述べ、分散および複製されたデータの扱いが、その引用仕様によって呼出し元でどのように仮定されるかを定義する。これにより、プログラマは、HPF 以外の言語の機能を利用できるようになる。例えば、HPF では効率的に扱えない問題に対しての低レベル機能の利用、重要なカーネル部分の人手チューニング、および最適化されたライブラリの呼出しなどが可能となる。このインタフェースは、C のような他言語と HPF とのインタフェースとしても利用できる。

6.1　概要

ある HPF プログラムから、他のプログラミングモデルや異なる言語で実装された手続を呼び出したい場合があるかもしれない。手続の**プログラミングモデル** (*programming model*) には、以下のようなものが与えられる可能性がある。

- 単一制御スレッドで、概念上は手続の一つのコピーが実行され、プログラムテキストには制御の単一の軌跡が存在するもの。このモデルは前提となるターゲットハードウェアが (潜在的に) 複数のプロセッサあるいはメモリを持っている場合にグローバル *(global)* と呼ばれ、前提となるターゲットハードウェアが単一プロセッサ (あるいはマルチプロセッサの中の 1 つのノード) である場合に、シリアル (*serial*) と呼ばれる。

- 複数制御スレッド (プロセッサ毎に 1 つ) で、それぞれのスレッドが同じ手続を実行するもの。このモデルは ローカル (*local*)、より一般的には SPMD(Single Program, Multiple Data) と呼ばれる。

- ここでは論じない、何等かの他のモデル。例えば、複数制御スレッドであり、プロセッサに対するループ繰返しの動的割り当てや、明示的なプロセス生成を備えているような

物である。呼出時の最初の時点で、概念上実行される手続のコピーが一つあるが、このコピーはプログラムテキスト内に複数の制御の軌跡を (おそらくは時間とともに数を変えながら) 生み出す可能性がある。

プログラミング言語 (*programming language*) は、ある特定の文法 (言語仕様)、語義 (意味)、及び実用性 (目的) を提供する。プログラミング言語の中には、Fortran(ANSI 及び ISO 規格。最新版は 1997 年に認可されるはずである)、HPF(Fortran に拡張と制約を加えた仕様)、Fortran 77(以前の ANSI 及び ISO 規格)、C 、C++ 、Java、Visual Basic、 そして COBOL が含まれる。

プログラム単位の言語とモデルは、全体でその外来種別 (*extrinsic kind*) を構成する。この外来種別は、外来プレフィックス (*extrinsic prefix*) によって明示的に、またはコンパイラの選択やある一連のコンパイラオプションによる起動によって暗黙に指定することができる。従って、コンパイラを Fortran で定義されている**親有効域** (*host scoping unit*) を与えるものとしてみることができる。例えば、HPF コンパイラでコンパイルされたあるプログラムは、`HPF` という外来種別に属することになる。また、プログラムの外来種別は `EXTRINSIC(HPF)` あるいは `EXTRINSIC(LANGUAGE='HPF',MODEL='GLOBAL')` のように明示的に外来プレフィックスを使用して指定することができる。

6.2 外来プログラム単位の宣言

6.2.1 FUNCTION 文及び SUBROUTINE 文

外来プレフィックス (*extrinsic-prefix*) は、(Fortran の規格で定義された) FUNCTION 文、及び SUBROUTINE 文の中のキーワード `RECURSIVE`、`PURE`、及び `ELEMENTAL` と同じ位置に記述することができる。これは、1995 年 5 月の Fortran95 規格のドラフトの中で、*prefix-spec* に関する規則 R1219 の拡張で規定されている。FUNCTION 文 (*function-stmt*) に関する規則 R1217、*prefix* に関する規則 R1218、及び SUBROUTINE 文 (*subroutine-stmt*) に関する規則 R1222 には変更はないが、参考のためにここに再掲する。

H601 *function-stmt* **is** [*prefix*] `FUNCTION` *function-name*
([*dummy-arg-name-list*])
[`RESULT` (*result-name*)]

H602 *subroutine-stmt* **is** [*prefix*] `SUBROUTINE` *subroutine-name*
[([*dummy-arg-list*])]

H603 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

H604 *prefix-spec* **is** *type-spec*
or `RECURSIVE`
or `PURE`
or `ELEMENTAL`
or *extrinsic-prefix*

制約: 任意の HPF の外部副プログラム内で、どの内部副プログラムも、その親と同じ外来種

別でなければならず、また、明示的に外来種別が与えられていない内部副プログラムに対しては、その親と同じ外来種別が仮定される。

F95:12.2 で与えられている**手続の特性**の定義を拡張して、手続の外来種別を含むものとする。

6.2.2 PROGRAM 文、MODULE 文、及び BLOCK DATA 文

外来プレフィックス (*extrinsic-prefix*) は、PROGRAM 文、MODULE 文、及び BLOCK DATA 文の先頭にも記述することができる。以下の構文は、Fortran95 の構文規則 R1102(PROGRAM 文 (*program-stmt*))、R1105(MODULE 文 (*module-stmt*))、及び R1111(BLOCK DATA 文 (*block-data-stmt*)) の拡張である。

H605 *program-stmt* **is** [*extrinsic-prefix*] `PROGRAM` *program-name*

H606 *module-stmt* **is** [*extrinsic-prefix*] `MODULE` *module-name*

H607 *block-data-stmt* **is** [*extrinsic-prefix*] `BLOCK DATA`
[*block-data-name*]

制約: 任意の HPF のモジュール内で、どのモジュール副プログラムも、その親と同じ外来種別でなければならず、また明示的に外来種別が与えられていないモジュール副プログラムに対しては、その親と同じ外来種別が仮定される。

制約: HPF の任意の主プログラムあるいはモジュール副プログラム内で、どの内部副プログラムも、その親と同じ外来種別でなければならず、また明示的に外来種別が与えられていない内部副プログラムに対しては、その親と同じ外来種別が仮定される。

6.2.3 外来プレフィックス

H608 *extrinsic-prefix* **is** `EXTRINSIC (` *extrinsic-spec* `)`

H609 *extrinsic-spec* **is** *extrinsic-spec-arg-list*
or *extrinsic-kind-keyword*

H610 *extrinsic-spec-arg* **is** *language*
or *model*
or *external-name*

H611 *language* **is** [`LANGUAGE =`]
scalar-char-initialization-expr

H612 *model* **is** [`MODEL =`]
scalar-char-initialization-expr

H613 *external-name* **is** [`EXTERNAL_NAME =`]
scalar-char-initialization-expr

制約: *extrinsic-spec-arg-list* の中には、*language*、*model*、あるいは *external-name* の少なくとも 1 つが指定されなければならず、またどれも 2 回以上指定することはできない。

制約: もし、LANGUAGE=を使わずに *language* を指定する場合、*language* は *extrinsic-spec-arg-list* 中の最初の要素でなければならない。もし、MODEL=を使わずに *model* を指定する場合、LANGUAGE=のない *language* が *extrinsic-spec-arg-list* 中の最初の要素であり、*model* が 2 番目の要素でなければならない。もし、EXTERNAL_NAME=を使わずに *external-name* を指定する場合、LANGUAGE=のない *language* が *extrinsic-spec-arg-list* 中の最初の要素であり、MODEL=のない *model* が 2 番目の要素でなければならない。

制約: LANGUAGE=、MODEL=、EXTERNAL_NAME=を伴う形は、上で禁止された場合を除いて、いかなる順番で書いても構わない。

これらの *extrinsic-spec-arg-list* に関する規則は、あたかも EXTRINSIC が、LANGUAGE、MODEL、EXTERNAL_NAME という、それぞれが OPTIONAL であるような *dummy-arg-list* を使用した明示的引用仕様を伴った手続であるかのようなものであることに注意されたい。

制約: *language* の中では、*char-initialization-expr* の値は、以下のものが許される。

- 'HPF'　HPF 言語を指す。もし *model* が明示的に指定されていない場合、*model* には'GLOBAL'が暗黙に仮定される。
- 'FORTRAN' ANSI または ISO 規格の Fortran 言語を指す。もし *model* が明示的に指定されていない場合、*model* には'SERIAL'が暗黙に仮定される。
- 'F77' 以前の ANSI または ISO 規格である FORTRAN 77 言語を指す。もし *model* が明示的に指定されていない場合、*model* には'SERIAL'が暗黙に仮定される。
- 'C' ANSI 規格の C 言語を指す。もし *model* が明示的に指定されていない場合、*model* には'SERIAL'が暗黙に仮定される。
- 実装依存の値。暗黙の *model* は実装に依存する。

ほとんどの実装にとって、'C'は、引用仕様本体 (*interface-body*) 中に記述された FUNCTION 文か SUBROUTINE 文でしか許されないことに注意されたい。

制約: *language* が指定されていない場合、親有効域と同じものが仮定される。

制約: *model* において、*char-initialization-expr* の値は、以下のものが許される。

- 'GLOBAL' グローバルモデルを指す。
- 'LOCAL' ローカルモデルを指す。
- 'SERIAL' シリアルモデルを指す。
- 実装依存の値。

制約: *model* が指定されず、*language* の指定から暗黙に仮定されない場合、親有効域と同じものが仮定される。

制約: 名称が HPF の 3 文字から始まる全ての *language* 及び *model* は、本仕様及びその後継仕様の現在あるいは将来における定義のために予約されている。

制約: *external-name* において、*scalar-char-initialization-expr* の値は、その用途が外来種別によって決定される文字列である。例えば、ある外来種別は、*external-name* を、その手続が C の手続から参照された場合の名前を指定するために使用するかもしれない。そのような実装では、ユーザはコンパイラに、その名前が C コンパイラに理解できるように変換を行なうことを期待するであろう。もし *external-name* が指定されていない場合、その値は実装依存となる。

H614 *extrinsic-kind-keyword* **is** HPF
or HPF_LOCAL
or HPF_SERIAL

制約: EXTRINSIC(HPF) は EXTRINSIC('HPF','GLOBAL') と同値である。*extrinsic-prefix* が存在しないとき、HPF コンパイラはコンパイル単位を外来種別 HPF に属するかのように解釈する。従って、HPF コンパイラにとって、EXTRINSIC(HPF) あるいは EXTRINSIC('HPF', GLOBAL') と指定するのは冗長である。しかし、このような明示的な指定は、複数の外来種別をサポートしているコンパイラを使用する場合に必要となるかもしれない。

制約: EXTRINSIC(HPF_LOCAL) は EXTRINSIC('HPF','LOCAL') と同値である。外来種別が HPF_LOCAL であるような主プログラムは、外来種別が HPF で、実行部がそのサブルーチンの呼出しだけから構成された主プログラムから引数なしで呼び出される、外来種別 HPF_LOCAL のサブルーチンであるかのように振る舞う。

制約: EXTRINSIC(HPF_SERIAL) は EXTRINSIC('HPF','SERIAL') と同値である。外来種別が HPF_SERIAL であるような主プログラムは、外来種別が HPF で、実行部分がそのサブルーチンの呼出しだけから構成された主プログラムから引数なしで呼び出される、外来種別 HPF_SERIAL のサブルーチンであるかのように振る舞う。

制約: 名称が HPF の 3 文字から始まる全ての *extrinsic-kind-keyword* は、本仕様及びその後継仕様の現在あるいは将来における定義のために予約されている。

【実装者への助言】

他の *language* あるいは *model* を、コンパイラベンダが定義、提供することができる。それらは、この HPF 仕様の一部ではないとはいえ、HPF の外来種別の規則と精神に添うものであるよう求められる。

実装において、プログラマに対するある程度の制約を課すことがあるかもしれない。さらに、それぞれの外来種別が、一連の異なる制約を求めることがありうる。

例えば、並列プロセッサでの実装では、スカラ引数を複製して全てのプロセッサにコピーを提供するほうが便利な場合がある。これは、このプロセスが呼出し元に見えない限り許される。これを達成するための方法の一つは、副プログラムから戻るときに、該当のスカラ引数の全てのコピーが同じ値を持っていなければならないという制約をプログラマに課すことである。これは、仮引数が INTENT(OUT) 属性を持っていれば、副プログラムの戻りの時点までには全てのコピーが矛盾なく更新されていなければならないということを暗黙に意味している。【以上】

6.3 HPF 外来副プログラムの呼出し

外来手続の呼出しは、HPF で記述されている呼出し側プログラムから見て、副プログラムが HPF で記述されている場合と厳密に同じように振る舞う。HPF の外来種別を持つプログラム単位から呼び出された関数またはサブルーチンが、呼出し元からみて明示的引用仕様を持たない場合、呼出し元と同じ外来種別を持つと仮定される。

呼出し元と異なる外来種別を持つ副プログラムを呼び出すためには、副プログラムは、呼出し元に認識できる明示的引用仕様を持たねばならない。そして副プログラムは、呼出し元から見て、呼出し元と同じ外来種別のコードで記述された場合とおおむね同じように振る舞うことが期待される。この要請を満たすための責任のいくらかはコンパイラにかかっており、またいくらかはプログラマにかかっている。この引用仕様は外来手続の「HPF への見え方」を定義する。

HPF 以外のモデル、あるいは言語で記述された手続は、ローカル手続実行モデルを使用しているか否かにかかわらず、その手続を呼び出す HPF プログラムの中で、`EXTRINSIC` として宣言されていなければならない。外来プレフィックスは、指定された副プログラムを呼び出すときにどの種類のインタフェースを使用するかを宣言する。もし明示的な指定がなければ、ユーザは実装依存のインタフェースの正当性について全ての責任を負わねばならない。

HPF の外来種別をもつプログラム単位中の引用仕様宣言に現れる FUNCTION 文または SUBROUTINE 文は、言語の実装によってサポートされている任意の外来種別に対する外来プレフィックスを持つことができる。その様な FUNCTION 文または SUBROUTINE 文に、外来プレフィックスが現れない場合、引用仕様宣言が現れたプログラムと同じ HPF の外来種別であることが仮定される。

引用仕様本体によって定義されている手続の特性は、手続の定義におけるそれと一致していなければならない。

外来引用仕様を持つ手続についての定義および規則は、HPF の範疇外である。しかし、その様な手続に対する明示的引用仕様は、HPF の仕様に従っていなければならない。個々の HPF の実装は、外来種別をどのように選択してサポートしてもよく、また全く何もサポートしなくてもかまわない (HPF の実装において明らかにサポートされなければならない外来種別「`HPF`」自体を除いて)。

6.3.1 型、手続、データに対するアクセス

一般に、ある外来種別を与えられたプログラム単位は、Fortran の有効域の規則に従って、同じ外来種別を持つ他のプログラム単位中の、型、手続、データの名前を使用することができる。

異なる外来種別を持つ他のプログラム単位中の型、手続、データの名前を使用することには、表 6.1 に示される、以下で述べるような制限が加えられる。

外来種別 HPF を持つあるモジュール X が、他の外来種別 HPF を持つプログラム単位 Y で使用されている場合、X の項目の内で、Y が使用または呼び出すことを許可されたものの名前だけが参照結合できることに注意されたい。すなわち、Y が使用できないすべての項目については X が非公開属性とするか、Y の `USE` 文が、Y が使用可能な項目の名前だけを並べた `ONLY` 句を持っているかの何れかでなければならない。

		使用されるモジュールの外来種別		
		HPF	HPF_SERIAL	HPF_LOCAL
使用する	HPF	**T P D**	**T P**	**T P**
プログラムの	HPF_SERIAL	**T**	**T P D**	**T**
外来種別	HPF_LOCAL	**T**	**T**	**T P D**

T = 構造型の定義
P = 手続及び手続の引用仕様
D = データ実体

表 6.1: それぞれの HPF 外来種別に従って、モジュールからの引用が可能な言語要素

6.3.1.1 型

明示的にマップされた成分を持たない構造型の定義は「外来種別なし (extrinsic kind neutral)」と考えることができる。外来種別 HPF を持つ任意のプログラム単位は、外来種別 HPF を持つどのようなモジュールからでも構造型の定義を使用することができる。公認拡張では、構造型の成分のマッピングが許可されていることに注意されたい。

6.3.1.2 手続

HPF のグローバルなプログラムまたは手続は、グローバル、ローカル、あるいはシリアルな他の HPF 手続を呼び出すことができる。

HPF のローカルなプログラムまたは手続は、他のローカルな HPF 手続だけを呼び出すことができる。グローバルまたはシリアルな手続は呼び出せない。

HPF のシリアルなプログラムまたは手続は、他のシリアルな HPF 手続だけを呼び出すことができる。グローバルまたはローカルな手続は呼び出せない。

6.3.1.3 データ

HPF 種別を持つプログラム単位の任意の名前つき共通ブロック (COMMON) は、もし存在すれば、同じ外来種別を持つ他の全てのプログラム単位の同じ名前の共通ブロックと結合する。無名共通ブロックも同様である。(このような共通ブロックの記憶領域は該当の外来種別を持つプログラムの中で宣言されたデータ実体と同じ振る舞いをする。特に、HPF_LOCAL なプログラムでは、各プロセッサに共通ブロックのコピーが存在することになるであろう。)

どのような共通ブロック名も、単一のプログラム中で異なる HPF 種別を持つプログラム単位内で使用されてはならない。同様に、単一のプログラム中で異なる HPF 種別を持つプログラム単位内で、無名共通ブロックを使用することはできない。

6.3.2 呼出しの効果

外来手続の呼出しは、その引数を再マップしない通常の HPF 手続の呼出しと意味的に同じでなければならない。従って、外来手続の呼出しは、以下のように動作したかの**如くに** (*as if*)

振る舞わなければならない。HPF の技術用語で**如くに** (*as if*) とは、記述された動作が、あたかもそれが指定された順序で実際に行なわれたかのようにユーザには見える、ということを意味する。実装では、ユーザに正しい結果を提供するように、任意の動作を任意の順序で行ってかまわない。

1. 副プログラムの呼出しに先行する呼出し元の全ての動作は、副プログラムの実行のいかなる動作よりも前に完了していなければならない。また、副プログラムの全ての動作は、呼出し元の副プログラムの呼出しに後続するいかなる動作よりも前に完了していなければならない。
2. 全ての実引数は、必要ならば、外来手続の引用仕様宣言の (明示的あるいは暗黙の) 指示に従って再マップされる。従って、引用仕様に現れる HPF のマッピング指示は拘束力がある。コンパイラは、ローカル外来手続の呼出しにおいてこれらの指示に従わなければならない。非外来副プログラムの場合と同様に、実引数はどのようにマップされていてもかまわない。もし必要なら、それらは呼出しの前に正しくマップされた一時領域に自動的にコピーされ、外来手続の戻りの前に実引数にコピーバックされる。スカラ仮引数やスカラ関数の戻り値は、各プロセッサで複製されたかのように振る舞う。これらのマッピングは、引用仕様中に明示されてもかまわないが、それ以外のいかなる明示的マッピングも、HPF 規格合致ではない。
3. `IN`、`OUT`、及び `INOUT` の授受特性の制約は、守られなければならない。
4. HPF 変数 (HPF variable) は、同じ明示的引用仕様を持つ HPF 手続によって更新されることはありうるが、それ以外は更新されない。`HPF_LOCAL` および `HPF_SERIAL` 手続は HPF のグローバルデータを参照することも更新することもできないが、他の種類の外来手続は HPF 手続と同程度にそれが可能かもしれないことに注意されたい。
5. ある手続から戻り、呼出し元が実行を再開するとき、呼出しの後に呼出し元が使用できる全ての実体は呼出し前の状態に正確にマップされていなければならない。特に、必要であれば、引数のもとのマッピングは復元される。
6. 副プログラムの呼出しの前後では、HPF の実行環境から見て厳密に同じプロセッサ集合になる。

【実装者への助言】

呼出しよりも論理的に先行する動作の完了を保証するために、複数のプロセッサが呼出しの前に同期処理を行う必要があるかもしれない。

呼び出された手続で使用可能な変数のマッピングが複製である場合、全てのコピーは、呼出しに先だって、原始プログラムの逐次的な意味に従った正確な値に更新されなければならないかもしれない。

複製された変数が、もし手続の中で更新されているならば、矛盾しないように更新されなければならない。より正確には、ある手続が使用可能な変数のマッピングが複製であり、(一つのあるいは複数の複製された) 手続によって更新された場合、最後のプロセッサがローカル手続から戻った時点でその複製された変数の全てのコピーは同じ値を持っていなければならない。

実装においては、ローカルな副プログラムの戻りの前に、複製された変数が副プログラムによって矛盾なく更新されているか判定するかも知れない。しかし、実装でそれが要求されているわけではないことに注意されたい。これは単に、例えば実行性能とデバッグの容易性との兼ね合いである。

グローバル HPF の副プログラムにおいては、副プログラムからの戻り時にその効果が取り消される限り、実引数はどのように複製あるいは再マップされてもよいことに注意されたい。

呼出し元の処理が再び実行されるよりも前に、手続の全ての動作の論理的な完了を保証するために、複数のプロセッサが呼出しの後で同期処理を行わなければならないかもしれない。

【以上】

6.4 外来手続の例

以下の例について考える。

```
PROGRAM DUMPLING
  INTERFACE
    EXTRINSIC('HPF','LOCAL') SUBROUTINE GNOCCHI(P, L, X)
      INTERFACE
        SUBROUTINE P(Q)
          REAL Q
        END SUBROUTINE P
        EXTRINSIC('COBOL','LOCAL') SUBROUTINE L(R)
          REAL R(:,:)
        END SUBROUTINE L
      END INTERFACE
      REAL X(:)
    END SUBROUTINE GNOCCHI
    EXTRINSIC('HPF','LOCAL') SUBROUTINE POTSTICKER(Q)
      REAL Q
    END SUBROUTINE POTSTICKER
    EXTRINSIC('COBOL','LOCAL') SUBROUTINE LEBERKNOEDEL(R)
      REAL R(:,:)
    END SUBROUTINE LEBERKNOEDEL
  END INTERFACE
  ...
  CALL GNOCCHI(POTSTICKER, LEBERKNOEDEL, (/ 1.2, 3.4, 5.6 /) )
  ...
END PROGRAM DUMPLING
```

主プログラム DUMPLINGは、HPF コンパイラでコンパイルされるときは、暗黙に外来種別 HPF に所属する。3 つの外来サブルーチン GNOCCHI、POTSTICKER、および LEBERKNOEDEL の引用仕様が宣言されている。最初の 2 つは外来種別 HPF_LOCAL に所属し、3 番目は COBOL_LOCAL に所属する。GNOCCHI は、2 つの仮手続を引数とすることができるので、引用仕様はそのための宣言をしなければならない。仮引数 P には、外来プレフィックスが与えられていないので、その外来種別は親有効域の外来種別になり、サブルーチン GNOCCHI の宣言から、それは外来種別 HPF_LOCAL を持つことになる。結合する実引数の宣言から、 POTSTICKER は明示的な外来プレフィックスを持つ必要がある。なぜならば、その親有効域はプログラム DUMPLING であり、外来種別 HPF に属しているからである。

もう少し例を挙げる。最初の例では、BAGEL の 100 という明示的な大きさの宣言は、そのグローバルな大きさを表し、ローカルな大きさではないことに注意されたい。

```
INTERFACE
  EXTRINSIC('HPF','LOCAL') FUNCTION BAGEL(X)
    REAL BAGEL(100)
    REAL X(:)
      !HPF$ DISTRIBUTE (CYCLIC) :: BAGEL, X
  END FUNCTION
END INTERFACE
```

次の例では、ALIGN 文により、X、Y、及び Z が全て同じ形状を持っていることに注意されたい。

```
INTERFACE OPERATOR (+)
  EXTRINSIC('C','LOCAL') FUNCTION LATKES(X, Y) RESULT(Z)
    REAL, DIMENSION(:,:), INTENT(IN) :: X
    REAL, DIMENSION(:,:), INTENT(IN) :: Y
    REAL, DIMENSION(SIZE(X,1), SIZE(X,2)) :: Z
      !HPF$ ALIGN WITH X :: Y, Z
      !HPF$ DISTRIBUTE (BLOCK, BLOCK) X
  END FUNCTION
END INTERFACE
```

次の最後の例での引用仕様宣言では、2 つの外部手続は、一方は外来手続で、もう一方はそうではないが、同じ総称名に結合されており、その配列引数と同じ型のスカラを戻す。

```
INTERFACE KNISH
  FUNCTION RKNISH(X)                          !通常の HPF の引用仕様
    REAL X(:), RKNISH
  END RKNISH
  EXTRINSIC('SISAL') FUNCTION CKNISH(X)   !外来引用仕様
    COMPLEX X(:), CKNISH
  END CKNISH
END INTERFACE
```

第7章　組込み手続とライブラリ手続

HPF の仕様には、Fortran の組込み手続が含まれるほか、2 つの範疇に属する新しい組込み手続が追加される。システム問合せ組込み関数と計算組込み関数である。

これら新しい組込み関数に加え、HPF では HPF_LIBRARY というライブラリモジュールも定義されており、HPF 仕様を完全に実装するベンダは必ずこれを提供しなければならない。

本章の HPF 組込み手続、ライブラリ手続に関する記述は、Fortran 規格の形式と規定に従っている。F95:13.1、F95:13.2、F95:13.3、F95:13.5.7、F95:13.8.1、F95:13.8.2、F95:13.9、及び F95:13.10 節の内容は、HPF の組込み手続やライブラリ手続、及びそれらに関する本章の解説に対しても有効である。

7.1　表記法

本章の例題において、T と F は、それぞれ論理値の真と偽を表すものとする。

7.2　システム問合せ組込み関数

マルチプロセッサ上の実装では、プロセッサの構成は実装ごとに異なる多次元プロセッサ配列となる。システム問合せ関数の返す値は、プロセッサ配列の大きさや形状といった、実際の計算機やプロセッサの構成に関するものである。NUMBER_OF_PROCESSORS は、プログラムで利用可能なプロセッサの総数、またはプロセッサ配列の指定された次元における、プログラムで利用可能なプロセッサの数を返す。また PROCESSORS_SHAPE は、プロセッサ配列の形状を返す。

Fortran の制限式の定義は、HPF のシステム問合せ組込み関数を引用できるよう拡張される。特に、Fortran 規格の 7.1.6.2 項のリストの最後に、以下の項目が追加される。

(13) システム問合せ関数 NUMBER_OF_PROCESSORS または PROCESSORS_SHAPE であって、あらゆる引数が制限式であるものの引用。

モジュールまたは主プログラムの有効域において、HPF 指示文中の制限式の中に現れる変数は、DO 型並び変数であるか、配列問合せ関数、ビット問合せ関数、文字問合せ関数、種別問合せ関数、数値問合せ関数のいずれかの引数としての引用でなければならない。

システム問合せ組込み関数の返す値は、1 つのプログラムの実行中は不変である。したがって、NUMBER_OF_PROCESSORS と PROCESSORS_SHAPE は、制限式である値を持ち、Fortran の他の任意の制限式が使用できるところならばどこでも使用できる。特に、NUMBER_OF_PROCESSORS は、宣言式中で使用できる。

システム問合せ関数の値を、初期化式中で引用してはならない。それらは定数ではないことがあるからである。HPF プログラムは、翻訳時には構成の分らない計算機向けに翻訳されることもある。

システム問合せ関数が問い合わせるのは物理的な計算機に関してであり、いかなるPROCESSORS 指示文とも無関係であることに注意されたい。HPF プログラムがより大きな計算機の物理的な一部分で実行されている場合には、問合せが行われるのは実際にプログラムを実行している小さな部分に関してであり、その諸元がシステム問合せ関数により返される。

関数 PROCESSORS_SHAPE の値として返すべき「自然な」形状を持たない計算機もある。たとえば、木構造のトポロジを持つ計算機などである。このような場合は、たとえば大きさが NUMBER_OF_PROCESSORS() の 1 次元配列など、ある程度の合理性と一貫性を持った計算機構成を与えるように実装されなければならない。言語処理系は、この構成と実際のハードウェア・プロセッサ識別方法との間の対応付けを行なう必要がある。

【利用者への助言】 SIZE(PROCESSORS_SHAPE()) は、プロセッサ配列の次元数を返す。配列の宣言や HPF 指示文中で、システム問合せ関数を以下のように引用することができる。

```
      INTEGER, DIMENSION(SIZE(PROCESSORS_SHAPE())) :: PSHAPE
!HPF$ TEMPLATE T(100, 3*NUMBER_OF_PROCESSORS())
```

【以上】

7.3 計算組込み関数

HPF には新たに計算組込み関数が 1 つ追加された。ILEN は、整数値を格納するのに必要なビット数を計算する。

7.4 ライブラリ手続

本章で述べるマッピング問合せサブルーチンと計算関数は、HPF ライブラリモジュール HPF_LIBRARY の中に用意されている。これらの手続を引用する際は、各有効域において適宜 USE 文を指定しなければならない。これらは組込みではない。

7.4.1 マッピング問合せサブルーチン

HPF はデータマッピング指示文を提供するが、それは本質的には言語処理系に対する助言にすぎない。マッピング問合せサブルーチンにより、プログラムは実行時に、配列が実際にどのようにマッピングされているかを知ることができる。正確なマッピングを知ることは、EXTRINSIC 副プログラムを呼び出すときに特に重要である。このような理由から、配列の計算機上での実際のマッピングを記述するマッピング問合せサブルーチンが HPF に含められた。手続の数を抑えるため、問合せ手続は省略可能な INTENT (OUT) 引数を持つサブルーチンとして構成されている。

7.4.2 ビット操作関数

HPF ライブラリには、3 つの要素別処理ビット操作関数が含まれる。LEADZ は、整数表現の先頭にある 0 であるビットの数を計算する。POPCNT は、整数中の 1 であるビットの数を数える。POPPAR は、整数中の 1 であるビットの奇偶を計算する。

7.4.3 配列集計関数

HPF には Fortran の SUM、ANY などの組込み関数と同様の演算を行う配列集計関数が追加された。新しい集計関数は、IALL、IANY、IPARITY 及び PARITY であり、それぞれ交換則と結合則を満たす 2 進演算 IAND、IOR、IEOR、及び .NEQV. に対応する。

関数の仕様の記述で「XXX 集計」という用語が使われているが、XXX は上述の 2 進演算のうちの 1 つであるものとする。例示によってこれらを定義する。array の全要素のうち mask の対応する要素が真であるものの IAND 集計は、以下のプログラムで result に計算される整数型スカラとする。

```
result = IAND_IDENTITY_ELEMENT
DO i_1 = LBOUND(array,1), UBOUND(array,1)
  ...
    DO i_n = LBOUND(array,n), UBOUND(array,n)
      IF ( mask(i_1,i_2,...,i_n) ) &
        result = IAND( result, array(i_1,i_2,...,i_n) )
    END DO
  ...
END DO
```

ここで n は array の次元数とし、IAND_IDENTITY_ELEMENT は全ビットが 1 である整数とする (整数のビット列としての解釈は、F95:13.5.7 で与えられる)。ほかの 3 つの集計演算も同様に定義される。IOR と IEOR の単位元は 0 である。.NEQV. の単位元は .FALSE. である。

7.4.4 配列集計拡散関数

XXX_SCATTER 関数は、配列集計関数を一般化したもので、要素の任意の部分集合を集計して結果の要素を構成することができる。このとき、結果の各要素に対応する配列の部分集合は、重複部分を持たない。HPF に 11 ある集計演算は、各々集計拡散関数の 1 つに対応しており、それとは別に COPY_SCATTER によって、既存の値を対応する部分集合中の任意の 1 つの値で上書きすることもできる。元の配列のどの要素が結果のどの要素と対応するかは、本節で述べる。それらの値を集計する方法は、7.7 節の各関数の仕様中で述べる。

これらの関数は以下のような一般形を持つ。

```
XXX_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
```

ただし、あとで述べるような特殊な場合を除く。XXX の値として許されるのは、ALL、ANY、COPY、COUNT、IALL、IANY、IPARITY、MAXVAL、MINVAL、PARITY、PRODUCT、及び SUM である。配列 ARRAY、MASK、及びすべての INDX は、形状適合するものとする。配列 INDX は整

数型であり、引数 INDX の数は BASE の次元数と同じでなければならない。引数 MASK は論理型の引数で、省略可能とする。COUNT_SCATTER を除き、ARRAY と BASE は同じ型の配列とする。COUNT_SCATTER については、ARRAY が論理型、BASE が整数型とする (ALL_SCATTER、ANY_SCATTER、COUNT_SCATTER、及び PARITY_SCATTER の引数 ARRAY は、論理型でなければならない。これらの関数は、省略可能な引数 MASK を持たない。Fortran 規格書にならい、これらの関数に対する引数 ARRAY は、7.7 節の仕様の中では MASK と記述している)。どの場合も、結果の配列は BASE と同じ型、種別型パラメタ、及び形状を持つ。

配列 INDX は ARRAY と同じ形状なので、ARRAY のあらゆる要素 a に対して、INDX にはそれぞれ対応する要素が存在する。n を BASE の次元数とするとき、各 $j = 1, 2, \ldots, n$ に対して、s_j を ARRAY の要素 a に対応する INDXj の要素の値とする。これらの添字により、ARRAY の要素 a に影響を受ける結果の要素が決定する。すなわち、添字 s_j のそれぞれに対して、BASE の対応する添字は、$b_j = s_j$+LBOUND(BASE, j) - 1 によって与えられる。

整数 b_j, $j = 1, \ldots, n$ は、BASE の要素を選択する添字となる。すなわち BASE($b_1, b_2, \ldots, b_n$) が選択される。BASE と結果とは形状適合するから、BASE の各要素に対して、対応する結果の要素が存在する。

したがって配列 INDX は、ARRAY のすべての要素から、結果と BASE の選択された要素への写像を定める。見方を変えると、この写像は、BASE の各要素 b と ARRAY の要素の集合 S とを対応づけている。

S が空のとき、BASE の要素 b に対応する結果の要素は、b と同じ値を持つ。

S が空でないとき、S の要素は要素 b とともに集計されて、結果の要素を構成する。集計拡散関数の詳細な仕様に、値を集計する方法が述べられている。例えば SUM_SCATTER において、S の要素が $a_1, \ldots, a_m$ のとき、BASE の要素 b に対応する結果の要素は、SUM((/$a_1, a_2, \ldots, a_m, b$/)) を評価した結果となる。

配列 INDX の要素の値は非負でなければならず、INDXj の要素の値は SIZE(BASE, j) 以下でなければならない。計算結果は、BASE の上下限の宣言の影響を受けず、BASE の形状だけに依存する。

スカラは任意の配列と形状適合するから、配列 INDX の位置に使用できる。このとき、選択される結果の要素は、1 つの超平面をなす。以下の例を参照されたい。

省略可能な最後の引数 MASK が指定されたとき、ARRAY の要素のうち、MASK が真であるような位置のものだけが演算に加わる。他のすべての ARRAY と配列 INDX の要素は無視され、結果のどの要素にも影響を与えない。

例えば、

配列 A が $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$、配列 B が $\begin{bmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & -8 & -9 \end{bmatrix}$、

配列 I1 が $\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 3 & 2 & 1 \end{bmatrix}$、配列 I2 が $\begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix}$

のとき、

SUM_SCATTER(A, B, I1, I2) は、$\begin{bmatrix} 14 & 6 & 0 \\ 8 & -5 & -6 \\ 0 & -8 & -9 \end{bmatrix}$ 、

SUM_SCATTER(A, B, 2, I2) は、$\begin{bmatrix} -1 & -2 & -3 \\ 30 & 3 & -3 \\ -7 & -8 & -9 \end{bmatrix}$ 、

SUM_SCATTER(A, B, I1, 2) は、$\begin{bmatrix} -1 & 24 & -3 \\ -4 & 7 & -6 \\ -7 & -1 & -9 \end{bmatrix}$ 、

SUM_SCATTER(A, B, 2, 2) は、$\begin{bmatrix} -1 & -2 & -3 \\ -4 & 40 & -6 \\ -7 & -8 & -9 \end{bmatrix}$

になる。

配列 A が $\begin{bmatrix} 10 & 20 & 30 & 40 & -10 \end{bmatrix}$ 、B が $\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$ 、IND が $\begin{bmatrix} 3 & 2 & 2 & 1 & 1 \end{bmatrix}$ のとき、SUM_SCATTER(A, B, IND, MASK=(A .GT. 0)) は、$\begin{bmatrix} 41 & 52 & 13 & 4 \end{bmatrix}$ となる。

7.4.5 配列累計関数

ベクトルの累計では、結果の各要素は、先行するベクトル要素の関数 (順累計のとき) であるか、または後続するベクトル要素の関数 (逆累計のとき) である。これらの関数により、配列及び部分配列上の累計演算が提供される。関数は以下のような一般形を持つ。

```
XXX_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
XXX_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
```

ただし、あとで述べるような特殊な場合を除く。XXX の値として許されるのは、ALL、ANY、COPY、COUNT、IALL、IANY、IPARITY、MAXVAL、MINVAL、PARITY、PRODUCT、及び SUM である。

順方向型 (prefix) と逆方向型 (suffix) の双方の手続に当てはまるとき、以下ではそれらを関数 YYYFIX と呼ぶ。

引数 DIM、MASK、SEGMENT、及び EXCLUSIVE は省略可能である。関数 COPY_YYYFIX は、引数 MASK や EXCLUSIVE を持たない。また関数 ALL_YYYFIX、ANY_YYYFIX、COUNT_YYYFIX、及び PARITY_YYYFIX は、引数 MASK を持たない。これらの関数に対する引数 ARRAY は論理型でなければならず、7.7 節の仕様の中では MASK と記述している。

引数 MASK 及び SEGMENT は論理型でなければならない。SEGMENT の形状は ARRAY と同じでなければならない。MASK は ARRAY と形状適合しなければならない。EXCLUSIVE は論理型スカラとする。DIM は整数型スカラであって、1 から ARRAY の次元数までの範囲の値であるものする。

結果の値. 結果は、ARRAY と同じ形状とする。また COUNT_YYYFIX を除き、ARRAY と同じ型及び種別型パラメタを持つ (COUNT_YYYFIX の結果は、基本整数型とする)。

どの関数も、仕様に記載された関数ごとに決まったやり方で選ばれる ARRAY の要素の値によって、結果のすべての要素が決まる。省略可能な引数は、結果の各要素に対する ARRAY の要素の選択方法に影響を与える。ここで選ばれた ARRAY の要素は、結果の

要素に寄与する、と呼ばれる。ある結果の要素に対して ARRAY のどの要素が寄与するのか、本節で完全に述べる。

ある結果の要素に対して ARRAY の要素が一つも選択されないとき、その結果の要素には仕様に記載された関数ごとに決まった既定値が設定される。

ある結果の要素 r に対して、ARRAY の対応する要素を a とする。以下の規則のいずれかにより無効とされないすべての ARRAY の要素が r に寄与する。

1. 関数が XXX_PREFIX であるとき、配列要素順序で a より後のどの ARRAY の要素も r には寄与しない。関数が XXX_SUFFIX であるとき、配列要素順序で a より前のどの ARRAY の要素も r には寄与しない。
2. 引数 DIM が指定されたとき、第 DIM 次元の添字を除く、少なくとも 1 つの添字が対応する a の添字と異なるならば、ARRAY の要素 z は r に寄与しない (DIM が省略されたとき、ARRAY、MASK、及び SEGMENT は、一時的に 1 次元配列と見なされるかのように、配列要素順序で処理される。DIM が指定されたときは、完全に独立な一群の累計演算が ARRAY の選ばれた次元に沿って実行される)。
3. 引数 MASK が指定されたとき、ARRAY の要素 z は、MASK の対応する要素が真のときだけ r に寄与する (MASK が偽の位置に対応する配列要素は、結果にはまったく寄与しない。しかし、それでも結果は、MASK が偽であるような位置も含め、すべての位置で定義される)。
4. 引数 SEGMENT が指定されたとき、ARRAY の要素 z は、以下の性質をすべて満たす ARRAY の要素 w (z 自身のこともある) が存在するならば、寄与しない。
   (a) 関数が XXX_PREFIX のとき、w が配列要素順序で z より前にはないが、a よりは前にある。または関数が XXX_SUFFIX のとき、w が配列要素順序で z より後にはないが、a よりは後にある。
   (b) 引数 DIM が指定されたとき、第 DIM 次元の添字を除く w のすべての添字が、a の対応する添字と同じである。
   (c) w に対応する SEGMENT の要素は、a に対応する SEGMENT の要素と異なる値を持つ (言い換えれば、z が寄与するのは、z から a までの列に対応する SEGMENT の要素に同じ値が並んでいるときだけである)。
5. 引数 EXCLUSIVE が指定されていて、真であるとき、a 自身は r に寄与しない。

これらの一般則から、以下のような重要な場合が導かれる。

場合 *(i)*: ARRAY が 1 次元配列のとき、XXX_PREFIX(ARRAY) の結果の要素 i は、ARRAY の最初の i 個の要素で決まる。また XXX_SUFFIX(ARRAY) の結果の要素 SIZE(ARRAY) $- i + 1$ は、ARRAY の最後の i 個の要素で決まる。

場合 *(ii)*: ARRAY が 2 次元以上の配列のとき、XXX_PREFIX(ARRAY) の結果の各要素の値は、ARRAY の対応する要素 a と、配列要素順序で a より前のすべての ARRAY の要素で決まる。XXX_SUFFIX については、ARRAY の対応する要素 a と、配列要素順序で a より後の ARRAY の要素で決まる。

場合 *(iii)*: XXX_PREFIX(ARRAY,MASK=MASK) の結果の各要素の値は、ARRAY の対応する要素 a と、配列要素順序で a より前のすべての ARRAY の要素で決まるが、このとき MASK の対応する要素が真のものだけが結果に寄与する。この制

限のために寄与する配列要素が一つも選ばれないとき、結果の要素には関数ごとの既定値が設定される。

場合 *(iv)*: XXX_PREFIX(ARRAY,DIM=DIM) の結果の各要素の値は、ARRAY の選ばれた要素、すなわち、ARRAY の対応する要素 a と、第 DIM 次元で a より前のすべての ARRAY の要素で決まる。たとえば、SUM_PREFIX(A(1:N,1:N), DIM=2) の結果の要素 (i_1, i_2) は、SUM(A(i_1,1 : i_2)) で計算できる。より一般的に、SUM_PREFIX(ARRAY, DIM) の場合、結果の要素 $i_1, i_2, \ldots, i_{DIM}, \ldots, i_n$ は、SUM(ARRAY($i_1, i_2, \ldots, :i_{DIM}, \ldots, i_n$)) で計算できる (最後の式の、$i_{DIM}$ の前のコロンに注目されたい)。

場合 *(v)*: ARRAY が 1 次元配列のとき、XXX_PREFIX(ARRAY, EXCLUSIVE=.TRUE.) の結果の要素 i は、ARRAY の最初の $i-1$ 個の要素で決まる。

場合 *(vi)*: 省略可能な引数は、任意の組合せで指定できる。

【利用者への助言】 偽から真、または真から偽へのすべての**変わり目**で、新しい区分が始まる。すなわち 1 つの区分は、同じ論理値が連続する最長部分列によって指定される。

```
(/T,T,T,F,T,F,F,F,T,F,F,T/)
  ----- - - ----- - --- -     7 つの区分
```

【以上】

【仕様の根拠】

既存のライブラリには、各区分の**開始点**を指定することにより範囲を定めるものがある。また、各区分の**終了点**を指定することにより範囲を定めるライブラリもある。それぞれのやり方には、それぞれの利点がある。順方向型の代りに逆方向型を実行する場合、この規定を変えるべきかどうか、という問題もある。HPF では、上述の対称的な表現を採用している。この表現の主な利点は以下の通りである。

(A) 対称的であり、同じ区分指定子を (開始点か終了点かの) 解釈を変えずに順方向型に対しても逆方向型に対しても使用できる。

(B) PARITY_PREFIX または PARITY_SUFFIX を引用することで、開始ビット表現や終了ビット表現を容易にこの形式に変換できる。このような方法は、言語処理系に認識させる標準的な表現法であると言えるだろう。

```
SUM_PREFIX(FOO,SEGMENT=PARITY_PREFIX(START_BITS))
SUM_PREFIX(FOO,SEGMENT=PARITY_SUFFIX(STOP_BITS))
SUM_SUFFIX(FOO,SEGMENT=PARITY_SUFFIX(START_BITS))
SUM_SUFFIX(FOO,SEGMENT=PARITY_PREFIX(STOP_BITS))
```

【以上】

例示 . 以下の例は、SUM_PREFIX に対する省略可能な引数の、可能な組合せをすべて説明したものである。SUM_YYYFIX に対する既定値は 0 とする。

場合 *(i)*: SUM_PREFIX((/1,3,5,7/)) は、$\begin{bmatrix} 1 & 4 & 9 & 16 \end{bmatrix}$ となる。

場合 *(ii)*: B が配列 $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$ のとき、SUM_PREFIX(B) は配列 $\begin{bmatrix} 1 & 14 & 30 \\ 5 & 19 & 36 \\ 12 & 27 & 45 \end{bmatrix}$ となる。

場合 *(iii)*: A が配列 $\begin{bmatrix} 3 & 5 & -2 & -1 & 7 & 4 & 8 \end{bmatrix}$ のとき、SUM_PREFIX(A, MASK = A .LT. 6) は、$\begin{bmatrix} 3 & 8 & 6 & 5 & 5 & 9 & 9 \end{bmatrix}$ となる。

場合 *(iv)*: B が配列 $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

のとき、SUM_PREFIX(B, DIM=1) は配列 $\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 12 & 15 & 18 \end{bmatrix}$

となり、SUM_PREFIX(B, DIM=2) は配列 $\begin{bmatrix} 1 & 3 & 6 \\ 4 & 9 & 15 \\ 7 & 15 & 24 \end{bmatrix}$ となる。

場合 *(v)*: SUM_PREFIX((/1,3,5,7/), EXCLUSIVE=.TRUE.) は、$\begin{bmatrix} 0 & 1 & 4 & 9 \end{bmatrix}$ となる。

場合 *(vi)*: B が配列 $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix}$、M が配列 $\begin{bmatrix} \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} \\ \mathrm{F} & \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{T} \\ \mathrm{T} & \mathrm{F} & \mathrm{T} & \mathrm{F} & \mathrm{F} \end{bmatrix}$、S が配列 $\begin{bmatrix} \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{F} \\ \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{F} \\ \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} & \mathrm{T} \end{bmatrix}$ のとき、

SUM_PREFIX(B, DIM=2, MASK=M, SEGMENT=S, EXCLUSIVE=.TRUE.) は、$\begin{bmatrix} 0 & 1 & 0 & 3 & 7 \\ 0 & 0 & 0 & 0 & 9 \\ 0 & 11 & 11 & 24 & 24 \end{bmatrix}$ となる。

SUM_PREFIX(B, DIM=2, MASK=M, SEGMENT=S, EXCLUSIVE=.FALSE.) は、$\begin{bmatrix} 1 & 3 & 3 & 7 & 12 \\ 0 & 0 & 8 & 9 & 19 \\ 11 & 11 & 24 & 24 & 24 \end{bmatrix}$ となる。

SUM_PREFIX(B, DIM=2, MASK=M, EXCLUSIVE=.TRUE.) は、$\begin{bmatrix} 0 & 1 & 3 & 6 & 10 \\ 0 & 0 & 0 & 8 & 17 \\ 0 & 11 & 11 & 24 & 24 \end{bmatrix}$ となる。

SUM_PREFIX(B, DIM=2, MASK=M, EXCLUSIVE=.FALSE.) は、$\begin{bmatrix} 1 & 3 & 6 & 10 & 15 \\ 0 & 0 & 8 & 17 & 27 \\ 11 & 11 & 24 & 24 & 24 \end{bmatrix}$ となる。

SUM_PREFIX(B, DIM=2, SEGMENT=S, EXCLUSIVE=.TRUE.) は、$\begin{bmatrix} 0 & 1 & 0 & 3 & 7 \\ 0 & 0 & 7 & 0 & 9 \\ 0 & 11 & 23 & 36 & 50 \end{bmatrix}$

となる。

SUM_PREFIX(B, DIM=2, SEGMENT=S, EXCLUSIVE=.FALSE.) は、$\begin{bmatrix} 1 & 3 & 3 & 7 & 12 \\ 6 & 7 & 15 & 9 & 19 \\ 11 & 23 & 36 & 50 & 65 \end{bmatrix}$ となる。

SUM_PREFIX(B, DIM=2, EXCLUSIVE=.TRUE.) は、$\begin{bmatrix} 0 & 1 & 3 & 6 & 10 \\ 0 & 6 & 13 & 21 & 30 \\ 0 & 11 & 23 & 36 & 50 \end{bmatrix}$ となる。

SUM_PREFIX(B, DIM=2, EXCLUSIVE=.FALSE.) は、$\begin{bmatrix} 1 & 3 & 6 & 10 & 15 \\ 6 & 13 & 21 & 30 & 40 \\ 11 & 23 & 36 & 50 & 65 \end{bmatrix}$ となる。

SUM_PREFIX(B, MASK=M, SEGMENT=S, EXCLUSIVE=.TRUE.) は、$\begin{bmatrix} 0 & 11 & 0 & 0 & 0 \\ 0 & 13 & 0 & 4 & 5 \\ 0 & 13 & 8 & 0 & 0 \end{bmatrix}$

となる。

SUM_PREFIX(B, MASK=M, SEGMENT=S, EXCLUSIVE=.FALSE.) は、$\begin{bmatrix} 1 & 13 & 3 & 4 & 5 \\ 0 & 13 & 8 & 13 & 15 \\ 11 & 13 & 21 & 0 & 0 \end{bmatrix}$ となる。

SUM_PREFIX(B, MASK=M, EXCLUSIVE=.TRUE.) は、$\begin{bmatrix} 0 & 12 & 14 & 38 & 51 \\ 1 & 14 & 17 & 42 & 56 \\ 1 & 14 & 25 & 51 & 66 \end{bmatrix}$ となる。

SUM_PREFIX(B, MASK=M, EXCLUSIVE=.FALSE.) は、$\begin{bmatrix} 1 & 14 & 17 & 42 & 56 \\ 1 & 14 & 25 & 51 & 66 \\ 12 & 14 & 38 & 51 & 66 \end{bmatrix}$ となる。

SUM_PREFIX(B, SEGMENT=S, EXCLUSIVE=.TRUE.) は、$\begin{bmatrix} 0 & 11 & 0 & 0 & 0 \\ 0 & 13 & 0 & 4 & 5 \\ 0 & 20 & 8 & 0 & 0 \end{bmatrix}$ となる。

SUM_PREFIX(B, SEGMENT=S, EXCLUSIVE=.FALSE.) は、$\begin{bmatrix} 1 & 13 & 3 & 4 & 5 \\ 6 & 20 & 8 & 13 & 15 \\ 11 & 32 & 21 & 14 & 15 \end{bmatrix}$

となる。

SUM_PREFIX(B, EXCLUSIVE=.TRUE.) は、$\begin{bmatrix} 0 & 18 & 39 & 63 & 90 \\ 1 & 20 & 42 & 67 & 95 \\ 7 & 27 & 50 & 76 & 105 \end{bmatrix}$ となる。

SUM_PREFIX(B, EXCLUSIVE=.FALSE.) は、$\begin{bmatrix} 1 & 20 & 42 & 67 & 95 \\ 7 & 27 & 50 & 76 & 105 \\ 18 & 39 & 63 & 90 & 120 \end{bmatrix}$ となる。

7.4.6　配列ソート 関数

HPF には、多次元配列をソートするための手続が含まれている。関数 SORT_UP と SORT_DOWN は、ソートされた配列を返す。関数 GRADE_UP と GRADE_DOWN は、ソート置換列を返す。配列は、与えられた次元に沿ってソートすることも、全体配列を配列要素順序で一列に並んでいるとみなしてソートすることもできる。主キーと副キーによるソートが簡便にできるように、ソート置換列関数は安定ソートを用いる。

7.5　組込み手続とライブラリ手続の総称名

以下のすべての組込み手続とライブラリ手続において、表示された引数の名前は、実引数としてキーワード形式を使用するときに、キーワードとして用いなければならない名前である。多くの引数キーワードは、Fortran の場合と同様に、それらの使用法を示唆するような名前を持つ。F95:13.11 節を参照されたい。

7.5.1　システム 問合せ組込み関数

NUMBER_OF_PROCESSORS(DIM)　実行プロセッサの数
　　省略可能な引数　DIM
PROCESSORS_SHAPE()　　実行プロセッサ配列の形状

7.5.2　マッピング問合せサブルーチン

```
HPF_ALIGNMENT(ALIGNEE, LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, &
        NCOPIES)
```
　　省略可能な引数　LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, NCOPIES
```
HPF_DISTRIBUTION(DISTRIBUTEE, AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, &
        PROCESSORS_SHAPE)
```
　　省略可能な引数　AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE
```
HPF_TEMPLATE(ALIGNEE, TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, &
        NUMBER_ALIGNED)
```
　　省略可能な引数　TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO,
        NUMBER_ALIGNED

7.5.3　ビット 操作関数

ILEN(I)	ビット長 (組込み)
LEADZ(I)	先頭の 0 ビットの数
POPCNT(I)	ビット 1 の数
POPPAR(I)	ビット 1 の奇偶

7.5.4 配列集計関数

IALL(ARRAY, DIM, MASK)　　ビット論理積集計
　　省略可能な引数　DIM, MASK
IANY(ARRAY, DIM, MASK)　　ビット論理和集計
　　省略可能な引数　DIM, MASK
IPARITY(ARRAY, DIM, MASK)　　ビット排他的論理和集計
　　省略可能な引数　DIM, MASK
PARITY(MASK, DIM)　　排他的論理和集計
　　省略可能な引数　DIM

7.5.5 配列集計拡散関数

ALL_SCATTER(MASK, BASE, INDX1 ..., INDXn)
ANY_SCATTER(MASK, BASE, INDX1, ..., INDXn)
COPY_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
　　省略可能な引数　MASK
COUNT_SCATTER(MASK, BASE, INDX1, ..., INDXn)
IALL_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
　　省略可能な引数　MASK
IANY_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
　　省略可能な引数　MASK
IPARITY_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
　　省略可能な引数　MASK
MAXVAL_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
　　省略可能な引数　MASK
MINVAL_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
　　省略可能な引数　MASK
PARITY_SCATTER(MASK, BASE, INDX1, ..., INDXn)
PRODUCT_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
　　省略可能な引数　MASK
SUM_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
　　省略可能な引数　MASK

7.5.6 配列累計関数

ALL_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
　　省略可能な引数　DIM, SEGMENT, EXCLUSIVE
ALL_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
　　省略可能な引数　DIM, SEGMENT, EXCLUSIVE
ANY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
　　省略可能な引数　DIM, SEGMENT, EXCLUSIVE
ANY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
　　省略可能な引数　DIM, SEGMENT, EXCLUSIVE

COPY_PREFIX(ARRAY, DIM, SEGMENT)

    省略可能な引数 DIM, SEGMENT

COPY_SUFFIX(ARRAY, DIM, SEGMENT)

    省略可能な引数 DIM, SEGMENT

COUNT_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, SEGMENT, EXCLUSIVE

COUNT_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, SEGMENT, EXCLUSIVE

IALL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

IALL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

IANY_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

IANY_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

IPARITY_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

IPARITY_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

MAXVAL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

MAXVAL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

MINVAL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

MINVAL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

PARITY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, SEGMENT, EXCLUSIVE

PARITY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, SEGMENT, EXCLUSIVE

PRODUCT_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

PRODUCT_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

SUM_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

SUM_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

    省略可能な引数 DIM, MASK, SEGMENT, EXCLUSIVE

7.5.7　配列ソート 関数

GRADE_DOWN(ARRAY,DIM)　　降順ソートによる添字の置換
　　省略可能な引数　DIM
GRADE_UP(ARRAY,DIM)　　昇順ソートによる添字の置換
　　省略可能な引数　DIM
SORT_DOWN(ARRAY,DIM)　　降順ソート
　　省略可能な引数　DIM
SORT_UP(ARRAY,DIM)　　昇順ソート
　　省略可能な引数　DIM

7.6　組込み手続の仕様

ILEN(I)

機能. 整数の 2 の補数表現のビット長より 1 小さい値を返す。

分類. 要素別処理関数

引数. I は整数型でなければならない。

結果の型 及び 型パラメタ. I と同じとする。

結果の値.　I が非負のとき、ILEN(I) の値は $\lceil \log_2(\mathtt{I}+1) \rceil$ とする。I が負のとき、ILEN(I) の値は $\lceil \log_2(-\mathtt{I}) \rceil$ とする。

例示. ILEN(4) = 3 である。また ILEN(-4) = 2 である。2**ILEN(N-1) により、N は 2 のべき乗に切り上げられ (ただし $\mathtt{N} > 0$ とする)、2**(ILEN(N)-1) により、N は 2 のべき乗に切り捨てられる。LEADZ と比較されたい。

結果の値は、I の 2 の補数表現の長さより **1 小さい**値である。4 の最も短い 2 の補数表現は、0100 である。先頭の 0 は、符号ビットとして必要である。3 ビットの 2 の補数表現 100 は -4 を表す。

NUMBER_OF_PROCESSORS(DIM)

省略可能な引数. DIM

機能. プログラムで利用可能なプロセッサの総数、又はプロセッサ配列の指定された次元のプログラムで利用可能なプロセッサ数を返す。

分類. システム問合せ関数

引数.

DIM (省略可能) 整数型スカラであって、$1 \leq \mathtt{DIM} \leq n$ の範囲の値でなければならない。
ここで n は、プロセッサ配列の次元数とする。

結果の型、型パラメタ 及び 形状. 基本整数型スカラとする。

結果の値. 結果の値は、実装依存のハードウェアプロセッサ配列の第 DIM 次元の寸法とする。DIM が省略されたときは、実装依存のハードウェアプロセッサ配列の要素の総数とする。結果は常に 0 より大きいものとする。

例示. 8192 台のプロセッサが 128×64 の矩形格子状に編成されたコンピュータでは、NUMBER_OF_PROCESSORS() の値は 8192 になる。また NUMBER_OF_PROCESSORS(DIM=1) の値は 128 になり、NUMBER_OF_PROCESSORS(DIM=2) の値は 64 になる。単一プロセッサのワークステーションでは、NUMBER_OF_PROCESSORS() の値は 1 になる。スカラのプロセッサ配列の次元数は 0 なので、引数 DIM を指定することはできない。

PROCESSORS_SHAPE()

機能. 実装依存のプロセッサ配列の形状を返す。

分類. システム問合せ関数

引数. なし

結果の型、型パラメタ 及び 形状. 基本整数型 1 次元配列で、実装依存のプロセッサ配列の次元数に等しい大きさとする。

結果の値. 結果の値は、実装依存のプロセッサ配列の形状とする。

例示. 2048 台のプロセッサがハイパーキューブ状に編成されたコンピュータでは、PROCESSORS_SHAPE() の値は [2,2,2,2,2,2,2,2,2,2,2] になる。8192 台のプロセッサが 128×64 の矩形格子状に編成されたコンピュータでは、PROCESSORS_SHAPE() の値は [128,64] になる。単一プロセッサのワークステーションでは、PROCESSORS_SHAPE() の値は [] になる (大きさ 0 の 1 次元配列)。

7.7 ライブラリ手続の仕様

ALL_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, SEGMENT, EXCLUSIVE

機能. MASK の DIM 番目の次元について、区分ごとに順方向の論理積累計を計算する。

分類. 変型関数

引数.

MASK　　論理型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、MASK の次元数とする。

SEGMENT (省略可能) 論理型でなければならず、MASK と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. MASK と同じとする。

結果の値. 結果の要素 r の値は、ALL((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された MASK の要素の集合である (空集合のこともある)。

例示. ALL_PREFIX((/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{T} & \mathrm{T} \end{bmatrix}$ となる。

ALL_SCATTER(MASK,BASE,INDX1, ..., INDXn)

機能. MASK の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の要素は、BASE の対応する要素及びその位置に拡散される MASK の要素がすべて真であるとき、かつこのときに限り、真となる。

分類. 変型関数

引数.

MASK　論理型でなければならず、スカラであってはならない。

BASE　MASK と同じ種別型パラメタの論理型でなければならず、スカラであってはならない。

INDX1,...,INDXn　整数型でなければならず、MASK と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、ALL((/$a_1, a_2, ..., a_m, b$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる MASK の要素とする。

例示. ALL_SCATTER((/ T, T, T, F /), (/ T, T, T /), (/ 1, 1, 2, 2 /)) は $\begin{bmatrix} \mathrm{T} & \mathrm{F} & \mathrm{T} \end{bmatrix}$ となる。

ALL_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, SEGMENT, EXCLUSIVE

機能. MASK の DIM 番目の次元について、区分ごとに逆方向の論理積累計を計算する。

分類. 変型関数

引数.

MASK　　論理型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、MASK の次元数とする。

SEGMENT (省略可能) 論理型でなければならず、MASK と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. MASK と同じとする。

結果の値. 結果の要素 r の値は、ALL((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された MASK の要素の集合である (空集合のこともある)。

例示.　ALL_SUFFIX((/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} \mathrm{F} & \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{T} \end{bmatrix}$ となる。

ANY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, SEGMENT, EXCLUSIVE

機能. MASK の DIM 番目の次元について、区分ごとに順方向の論理和累計を計算する。

分類. 変型関数

引数.

MASK　　論理型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、MASK の次元数とする。

SEGMENT (省略可能) 論理型でなければならず、MASK と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. MASK と同じとする。

結果の値. 結果の要素 r の値は、ANY((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された MASK の要素の集合である (空集合のこともある)。

例示.　ANY_PREFIX((/F,T,F,F,F/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{F} \end{bmatrix}$ となる。

ANY_SCATTER(MASK,BASE,INDX1, ..., INDXn)

機能. MASK の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の要素は、BASE の対応する要素またはその位置に拡散される MASK の要素の中に一つでも真があるとき、かつこのときに限り、真となる。

分類. 変型関数

引数.

MASK　論理型でなければならず、スカラであってはならない。

BASE　MASK と同じ種別型パラメタの論理型でなければならず、スカラであってはならない。

INDX1,...,INDXn 整数型でなければならず、MASK と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、ANY((/$a_1, a_2, ..., a_m, b$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる MASK の要素とする。

例示. ANY_SCATTER((/ T, F, F, F /), (/ F, F, T /), (/ 1, 1, 2, 2 /)) は $\begin{bmatrix} \mathrm{T} & \mathrm{F} & \mathrm{T} \end{bmatrix}$ となる。

ANY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, SEGMENT, EXCLUSIVE

機能. MASK の DIM 番目の次元について、区分ごとに逆方向の論理和累計を計算する。

分類. 変型関数

引数.

MASK　論理型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、MASK の次元数とする。

SEGMENT (省略可能) 論理型でなければならず、MASK と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. MASK と同じとする。

結果の値. 結果の要素 r の値は、ANY((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された MASK の要素の集合である (空集合のこともある)。

例示. ANY_SUFFIX((/F,T,F,F,F/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{F} \end{bmatrix}$ となる。

COPY_PREFIX(ARRAY, DIM, SEGMENT)

省略可能な引数. DIM, SEGMENT

機能. ARRAY の DIM 番目の次元について、区分ごとに順方向の複写累計を計算する。

分類. 変型関数

引数.

ARRAY　どの型でもよい。スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq \mathtt{DIM} \leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、a_1 とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の、配列要素順序による集合である。この集合は決して空集合にはならない。

例示. COPY_PREFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 1 & 1 & 1 & 4 & 4 \end{bmatrix}$ となる。

COPY_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

省略可能な引数. MASK

機能. MASK で選択される ARRAY の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の各要素は、その位置に拡散される ARRAY の要素のうちの 1 つに等しいか、またはそのような要素がないときは、BASE の対応する要素に等しい。

分類. 変型関数

引数.

ARRAY　どの型でもよい。スカラであってはならない。

BASE　ARRAY と同じ型及び同じ種別型パラメタでなければならない。

INDX1,...,INDXn 整数型でなければならず、ARRAY と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. 7.4.4 項に述べた方法で BASE の要素 b と関連付けられる ARRAY の要素の集合を S とする。

S が空のとき、BASE の要素 b に対応する結果の要素の値は、b と同じとする。

S が空でないとき、BASE の要素 b に対応する結果の要素の値は、S から 1 つの要素を選択した結果とする。HPF はこの選択方法を規定せず、実装依存とする。

例示. COPY_SCATTER((/ 1, 2, 3, 4 /), (/ 7, 8, 9 /), (/ 1, 1, 2, 2 /)) は [x, y, 9] となる。ここで x は集合 $\{1,2\}$ の要素の 1 つであり、y は $\{3,4\}$ の要素の 1 つである。

COPY_SUFFIX(ARRAY, DIM, SEGMENT)

省略可能な引数. DIM, SEGMENT

機能. ARRAY の DIM 番目の次元について、区分ごとに逆方向の複写累計を計算する。

分類. 変型関数

引数.

ARRAY　どの型でもよい。スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、a_m とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の、配列要素順序による集合である。この集合は決して空集合にはならない。

例示. COPY_SUFFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 3 & 3 & 3 & 5 & 5 \end{bmatrix}$ となる。

COUNT_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, SEGMENT, EXCLUSIVE

機能. MASK の DIM 番目の次元について、区分ごとに順方向の COUNT 累計を計算する。

分類. 変型関数

引数.

MASK　論理型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、MASK の次元数とする。

SEGMENT (省略可能) 論理型でなければならず、MASK と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. 基本整数型であって、MASK と同じ形状の配列とする。

結果の値. 結果の要素 r の値は、COUNT((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された MASK の要素の集合である (空集合のこともある)。

例示. COUNT_PREFIX((/F,T,T,T,T/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 0 & 1 & 2 & 1 & 2 \end{bmatrix}$ となる。

COUNT_SCATTER(MASK,BASE,INDX1, ..., INDXn)

機能. MASK の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の各要素は、BASE の対応する要素と、その位置に拡散される MASK 中の真である要素の個数の和となる。

分類. 変型関数

引数.

MASK 論理型でなければならず、スカラであってはならない。

BASE 整数型でなければならず、スカラであってはならない。

INDX1,...,INDXn 整数型でなければならず、MASK と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、b + COUNT((/$a_1, a_2, ..., a_m$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる ARRAY の要素とする。

例示. COUNT_SCATTER((/ T, T, T, F /), (/ 1, -1, 0 /), (/ 1, 1, 2, 2 /)) は $\begin{bmatrix} 3 & 0 & 0 \end{bmatrix}$ となる。

COUNT_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, SEGMENT, EXCLUSIVE

機能. MASK の DIM 番目の次元について、区分ごとに逆方向の COUNT 累計を計算する。

分類. 変型関数

引数.

MASK　　論理型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、MASK の次元数とする。

SEGMENT (省略可能) 論理型でなければならず、MASK と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. 基本整数型であって、MASK と同じ形状の配列とする。

結果の値.　結果の要素 r の値は、COUNT((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された MASK の要素の集合である (空集合のこともある)。

例示.　COUNT_SUFFIX((/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 2 & 1 & 1 & 2 & 1 \end{bmatrix}$ となる。

GRADE_DOWN(ARRAY,DIM)

省略可能な引数. DIM

機能. 要素の値の降順ソートを表現する、基底を 1 とした配列添字の置換列を生成する。

分類. 変型関数

引数.

ARRAY　　整数型、実数型又は文字型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。対応する実引数は、省略可能な仮引数であってはならない。

結果の型、型パラメタ 及び 形状. 基本整数型とする。DIM を指定したとき、ARRAY と同じ形状とする。DIM を省略したとき、(/ SIZE(SHAPE(ARRAY)), SIZE(ARRAY) /) という形状とする。

結果の値.

場合 *(i)*:　S = GRADE_DOWN(ARRAY)
+ SPREAD(LBOUND(ARRAY),DIM=2, NCOPIES=SIZE(ARRAY))-1
の結果は、大きさ SIZE(ARRAY) の 1 次元配列 B を
FORALL (K=1:SIZE(B)) B(K)=ARRAY(S(1,K),S(2,K),...,S(N,K))

によって計算したとき、Bが降順にソートされるような値とする。ここで、Nは SIZE(SHAPE(ARRAY)) とする。さらに、S のすべての列は異なるものとする。すなわち、$j \neq m$ ならば ALL(S(:,j) .EQ. S(:,m)) は偽となる。また、ソートは安定である。すなわち $j \leq m$ かつ B(j) = B(m) ならば、ARRAY の配列要素順序で ARRAY(S($1,j$),S($2,j$),...,S(n,j)) が ARRAY(S($1,m$),S($2,m$),...,S(n,m)) に先行する。文字型配列の大小順序は、Fortran の組込み関数で使われるもの、すなわち ASCII 大小順序とする。

場合 *(ii)*: R = GRADE_DOWN(ARRAY, DIM=K) + LBOUND(ARRAY, DIM=K) - 1
の結果は、配列 B を
B($i_1, i_2, \ldots, i_k, \ldots, i_n$) =
ARRAY($i_1, i_2, \ldots,$ R($i_1, i_2, \ldots, i_k, \ldots, i_n$), $\ldots, i_n$)
によって計算したとき、すべての $i_1, i_2, \ldots, (i_k$を除く$), \ldots, i_n$ についてベクトル B($i_1, i_2, \ldots, :, \ldots, i_n$) が降順にソートされるような値とする。さらに、R($i_1, i_2, \ldots, :, \ldots, i_n$) は 1 から SIZE(ARRAY, DIM=K) までのすべての整数を並べ替えたものとする。また、ソートは安定である。すなわち $j \leq m$ かつ B($i_1, i_2, \ldots, j, \ldots, i_n$) = B($i_1, i_2, \ldots, m, \ldots, i_n$) ならば、R($i_1, i_2, \ldots, j, \ldots, i_n$) $\leq$ R($i_1, i_2, \ldots, m, \ldots, i_n$) となる。文字型配列の大小順序は、Fortran の組込み関数で使われるもの、すなわち ASCII 大小順序とする。

例示.

場合 *(i)*: GRADE_DOWN((/30, 20, 30, 40, -10/)) は、形状が $\begin{bmatrix} 1 & 5 \end{bmatrix}$ の 2 次元配列で、値は $\begin{bmatrix} 4 & 1 & 3 & 2 & 5 \end{bmatrix}$ になる (結果を 1 次元配列にするためには、省略可能な引数 DIM = 1 を指定しなければならない)。

配列 A が $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$ であるとき、GRADE_DOWN(A) の値は、

$\begin{bmatrix} 1 & 2 & 2 & 3 & 3 & 1 & 2 & 1 & 3 \\ 2 & 2 & 1 & 3 & 2 & 3 & 3 & 1 & 1 \end{bmatrix}$ になる。

場合 *(ii)*: 配列 A が $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$

であるとき、GRADE_DOWN(A, DIM = 1) の値は $\begin{bmatrix} 2 & 1 & 3 \\ 1 & 2 & 1 \\ 3 & 3 & 2 \end{bmatrix}$ になり、

GRADE_DOWN(A, DIM = 2) の値は $\begin{bmatrix} 2 & 3 & 1 \\ 2 & 1 & 3 \\ 3 & 2 & 1 \end{bmatrix}$ になる。

GRADE_UP(ARRAY,DIM)

省略可能な引数. DIM

機能. 要素の値の昇順ソートを表現する、基底を 1 とした配列添字の置換列を生成する。

分類. 変型関数

引数.

ARRAY　整数型、実数型又は文字型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。対応する実引数は、省略可能な仮引数であってはならない。

結果の型、型パラメタ 及び 形状. 基本整数型とする。DIM を指定したとき、ARRAY と同じ形状とする。DIM を省略したとき、(/ SIZE(SHAPE(ARRAY)), SIZE(ARRAY) /) という形状とする。

結果の値.

場合 *(i)*: S = GRADE_UP(ARRAY)
+ SPREAD(LBOUND(ARRAY),DIM=2, NCOPIES=SIZE(ARRAY))-1
の結果は、大きさ SIZE(ARRAY) の 1 次元配列 B を
FORALL (K=1:SIZE(B)) B(K)=ARRAY(S(1,K),S(2,K),...,S(N,K))
によって計算したとき、Bが昇順にソートされるような値とする。ここで、N は SIZE(SHAPE(ARRAY)) とする。さらに、S のすべての列は異なるものとする。すなわち、$j \neq m$ ならば ALL(S(:,j) .EQ. S(:,m)) は偽となる。また、ソートは安定である。すなわち $j \leq m$ かつ B(j) = B(m) ならば、ARRAY の配列要素順序で ARRAY(S(1,j),S(2,j),...,S(n,j)) が ARRAY(S(1,m),S(2,m),...,S(n,m)) に先行する。文字型配列の大小順序は、Fortran の組込み関数で使われるもの、すなわち ASCII 大小順序とする。

場合 *(ii)*: R = GRADE_UP(ARRAY, DIM=K) + LBOUND(ARRAY, DIM=K) - 1
の結果は、配列 B を
B($i_1, i_2, \ldots, i_k, \ldots, i_n$) =
ARRAY($i_1, i_2, \ldots,$ R($i_1, i_2, \ldots, i_k, \ldots, i_n$), $\ldots, i_n$)
によって計算したとき、すべての $i_1, i_2, \ldots,$ (i_kを除く), $\ldots, i_n$ についてベクトル B($i_1, i_2, \ldots, :, \ldots, i_n$) が昇順にソートされるような値とする。さらに、R($i_1, i_2, \ldots, :, \ldots, i_n$) は 1 から SIZE(ARRAY, DIM=K) までのすべての整数を並べ替えたものとする。また、ソートは安定である。すなわち $j \leq m$ かつ B($i_1, i_2, \ldots, j, \ldots, i_n$) = B($i_1, i_2, \ldots, m, \ldots, i_n$) ならば、R($i_1, i_2, \ldots, j, \ldots, i_n$) $\leq$ R($i_1, i_2, \ldots, m, \ldots, i_n$) となる。文字型配列の大小順序は、Fortran の組込み関数で使われるもの、すなわち ASCII 大小順序とする。

例示.

場合 *(i)*: `GRADE_UP( (/30, 20, 30, 40, -10/) )` は、形状が $\begin{bmatrix} 1 & 5 \end{bmatrix}$ の 2 次元配列で、値は $\begin{bmatrix} 5 & 2 & 1 & 3 & 4 \end{bmatrix}$ になる (結果を 1 次元配列にするためには、省略可能な引数 `DIM = 1` を指定しなければならない)。

配列 `A` が $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$ であるとき、`GRADE_UP(A)` の値は、

$\begin{bmatrix} 1 & 3 & 3 & 1 & 2 & 2 & 3 & 2 & 1 \\ 1 & 1 & 2 & 3 & 3 & 1 & 3 & 2 & 2 \end{bmatrix}$ になる。

場合 *(ii)*: 配列 `A` が $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$

であるとき、`GRADE_UP(A, DIM = 1)` の値は $\begin{bmatrix} 1 & 3 & 1 \\ 3 & 2 & 2 \\ 2 & 1 & 3 \end{bmatrix}$ になり、

`GRADE_UP(A, DIM = 2)` の値は $\begin{bmatrix} 1 & 3 & 2 \\ 3 & 1 & 2 \\ 1 & 2 & 3 \end{bmatrix}$ になる。

HPF_ALIGNMENT(ALIGNEE, LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, NCOPIES)

省略可能な引数. LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, NCOPIES

機能. 変数と、それが最終的に整列する *align-target* (配列またはテンプレート) との対応に関する情報を返す。

分類. マッピング問合せサブルーチン

引数.

`ALIGNEE` どの型でもよく、スカラでも配列でもよい。大きさ引継ぎ配列であってはならない。結合変数グループのメンバであるとき、そのグループの結合被覆でなければならない (「結合変数グループ」及び「結合被覆」の定義については 3.8 項を参照されたい) [1]。空状態のポインタ又は割り付けられていない割付け配列であってはならない。`INTENT (IN)` 引数とする。

`ALIGNEE`がポインタであるときは、指示先の整列に関する情報が返されるものとする。指示先は大きさ引継ぎの仮引数又は大きさ引継ぎの仮引数の部分配列であってはならない。

[1] 訳注: 原文の誤り。HPF 2.0 では「結合被覆」の定義は抹消されている。`ALIGNEE` は結合変数グループのメンバであってはならないと解釈すべきである。

LB (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。ALIGNEE の i 番目の次元軸の最初の要素が、対応する *align-target* の次元軸の LB(i) 番目の要素に最終的に整列している。ALIGNEE の i 番目の次元軸が縮退しているとき、LB(i) に設定される値は実装依存とする。

UB (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。ALIGNEE の i 番目の次元軸の最後の要素が、対応する *align-target* の次元軸の UB(i) 番目の要素に最終的に整列している。ALIGNEE の i 番目の次元軸が縮退しているとき、UB(i) に設定される値は実装依存とする。

STRIDE (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。STRIDE の i 番目の要素には、ALIGNEE の i 番目の次元軸に沿って各要素が整列するときの刻み幅が設定される。ALIGNEE の i 番目の次元軸が縮退しているとき、STRIDE(i) は 0 に設定される。

AXIS_MAP (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。AXIS_MAP の i 番目の要素には、ALIGNEE の i 番目の次元軸に対応する *align-target* の次元軸が設定される。ALIGNEE の i 番目の次元軸が縮退しているとき、AXIS_MAP(i) は 0 に設定される。

IDENTITY_MAP (省略可能) 基本論理型スカラでなければならない。INTENT (OUT) 引数とする。ALIGNEE の最終 *align-target* が ALIGNEE と同じ形状を持ち、次元軸が恒等置換を使ってマップされ、刻み幅がすべて正である (したがって形状の制限から 1 に等しい) とき、真が設定される。それ以外のとき、偽が設定される。変数が、ALIGN 指示文や REALIGN 指示文の *alignee* に現れず、INHERIT 属性も持たないとき、IDENTITY_MAP は真に設定される。ただし、その他の状況でも真となることがある。

NCOPIES (省略可能) 基本整数型スカラでなければならない。INTENT (OUT) 引数とする。*align-target* に最終的に整列している ALIGNEE の複製の数が設定される。複製がないときは 1 が設定される。

ALIGNEE がスカラのとき、LB、UB、STRIDE、AXIS_MAP には何も設定されない。

例示. 以下の宣言が与えられ、

```
      REAL PI = 3.1415927
      DIMENSION A(10,10),B(20,30),C(20,40,10),D(40)
!HPF$ TEMPLATE T(40,20)
!HPF$ ALIGN A(I,:) WITH T(1+3*I,2:20:2)
!HPF$ ALIGN C(I,*,J) WITH T(J,21-I)
!HPF$ ALIGN D(I) WITH T(I,4)
!HPF$ PROCESSORS PROCS(4,2), SCALARPROC
```

```
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO PROCS
!HPF$ DISTRIBUTE B(CYCLIC,BLOCK) ONTO PROCS
!HPF$ DISTRIBUTE ONTO SCALARPROC :: PI
```

実際のマッピングが宣言通りに行なわれたとすると、HPF_ALIGNMENT の呼出しの結果は以下のようになる。

	A	B	C	D
LB	[4, 2]	[1, 1]	[20, N/A, 1]	[1]
UB	[31, 20]	[20, 30]	[1, N/A, 10]	[40]
STRIDE	[3, 2]	[1, 1]	[-1, 0, 1]	[1]
AXIS_MAP	[1, 2]	[1, 2]	[2, 0, 1]	[1]
IDENTITY_MAP	偽	真	偽	偽
NCOPIES	1	1	1	1

ここで、「N/A」は実装依存の結果を表わす。NCOPIES の利用法を次の例で見る。

```
      LOGICAL BOZO(20,20),RONALD_MCDONALD(20)
!HPF$ TEMPLATE EMMETT_KELLY(100,100)
!HPF$ ALIGN RONALD_MCDONALD(I) WITH BOZO(I,*)
!HPF$ ALIGN BOZO(J,K) WITH EMMETT_KELLY(J,5*K)
```

このとき、CALL HPF_ALIGNMENT(RONALD_MCDONALD, NCOPIES = NC) は NC に 20 を設定する。また次の例では、

```
      LOGICAL BOZO(20,20),RONALD_MCDONALD(20)
!HPF$ TEMPLATE WILLIE_WHISTLE(100)
!HPF$ ALIGN RONALD_MCDONALD(I) WITH BOZO(I,*)
!HPF$ ALIGN BOZO(J,*) WITH WILLIE_WHISTLE(5*J)
```

CALL HPF_ALIGNMENT(RONALD_MCDONALD, NCOPIES = NC) は NC に 1 を設定する。

HPF_DISTRIBUTION(DISTRIBUTEE, AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE)

省略可能な引数. AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE

機能. 変数の最終 *align-target* の分散に関する情報を返す。

分類. マッピング問合せサブルーチン

引数.

DISTRIBUTEE どの型でもよく、スカラでも配列でもよい。大きさ引継ぎ配列であってはならない。結合変数グループのメンバであるとき、そのグループの結合被覆でなければならない (「結合変数グループ」及び「結合被覆」の定義

については 3.8 項を参照されたい) [2]。空状態のポインタ又は割り付けられていない割付け配列であってはならない。INTENT (IN) 引数とする。

DISTRIBUTEE がポインタであるときは、指示先の分散に関する情報が返されるものとする。指示先は大きさ引継ぎの仮引数又は大きさ引継ぎの仮引数の部分配列であってはならない。

AXIS_TYPE (省略可能) 基本文字型の 1 次元配列でなければならない。長さは任意であるが、完全な値を得るためには少なくとも長さが 9 必要である。各要素には以下の値が文字組込み代入文と同じ方法で設定される。少なくとも DISTRIBUTEE が最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは HPF_TEMPLATE が TEMPLATE_RANK に返す値である)。INTENT (OUT) 引数とする。AXIS_TYPE の i 番目の要素には、*align-target* の i 番目の次元軸の分散に関する情報が設定される。HPF では以下の値が定義されている (実装によってはその他の値が定義されていることもある)。

'BLOCK' この次元軸は BLOCK 分散されている。AXIS_INFO の対応する要素にはブロックの大きさが設定される。

'COLLAPSED' この次元軸は縮退している (「*」指定により分散されている)。AXIS_INFO の対応する要素の値は実装依存とする。

'CYCLIC' この次元軸は CYCLIC 分散されている。AXIS_INFO の対応する要素にはブロックの大きさが設定される。

AXIS_INFO (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも DISTRIBUTEE が最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは HPF_TEMPLATE が TEMPLATE_RANK に返す値である)。INTENT (OUT) 引数とする。AXIS_INFO の i 番目の要素には、DISTRIBUTEE の最終 *align-target* の i 番目の次元軸の、ブロック分散又はサイクリック分散でのブロックの大きさが設定される。次元軸が縮退しているとき、設定される値は実装に依存する。

PROCESSORS_RANK (省略可能) 基本整数型スカラでなければならない。DISTRIBUTEE が分散されるプロセッサ構成の次元数が設定される。INTENT (OUT) 引数とする。

PROCESSORS_SHAPE (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも PROCESSORS_RANK の返す値 m に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。最初の m 個の要素に、DISTRIBUTEE がマップされるプロセッサ構成の形状が設定される (HPF_DISTRIBUTION は 2 度呼び出す必要があるかもしれない。すなわち最初の呼び出しで PROCESSORS_RANK の値を得、それにより PROCESSORS_SHAPE を割り付けるのである)。

例示. HPF_ALIGNMENT の例に示す宣言が与えられ、実際のマッピングが宣言通りに行なわれたとすると、HPF_DISTRIBUTE の呼出しの結果は以下のようになる。

[2] 訳注: 原文の誤り。HPF 2.0 では「結合被覆」の定義は抹消されている。DISTRIBUTEE は結合変数グループのメンバであってはならないと解釈すべきである。

	A	B	PI
AXIS_TYPE	['BLOCK', 'BLOCK']	['CYCLIC', 'BLOCK']	[ ]
AXIS_INFO	[10, 10]	[1, 15]	[ ]
PROCESSORS_SHAPE	[4, 2]	[2, 2]	[ ]
PROCESSORS_RANK	2	2	0

HPF_TEMPLATE(ALIGNEE, TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED)

省略可能な引数. LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED, TEMPLATE_RANK

機能. 変数の最終 *align-target* に関する情報を返す。HPF_ALIGNMENTが変数の観点からの情報を返すのに対し、HPF_TEMPLATEは最終 *align-target* の観点からの変数に関する情報を返す。

分類. マッピング問合せサブルーチン

引数.

ALIGNEE どの型でもよく、スカラでも配列でもよい。大きさ引継ぎ配列であってはならない。結合変数グループのメンバであるとき、そのグループの結合被覆でなければならない (「結合変数グループ」及び「結合被覆」の定義については 3.8 項を参照されたい) [3]。空状態のポインタ又は割り付けられていない割付け配列であってはならない。INTENT (IN) 引数とする。

ALIGNEEがポインタであるときは、指示先の整列に関する情報が返されるものとする。指示先は大きさ引継ぎの仮引数又は大きさ引継ぎの仮引数の部分配列であってはならない。

TEMPLATE_RANK (省略可能) 基本整数型スカラでなければならない。INTENT (OUT) 引数とする。最終 *align-target* の次元数が設定される。この値は、縮退や複製により ALIGNEE の次元数と異なることがある。

LB (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEEが最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは TEMPLATE_RANK に設定される値である)。INTENT (OUT) 引数とする。LB の i 番目の要素には、テンプレートの i 番目の次元軸に対する *align-target* の宣言時の下限が設定される。

UB (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEEが最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは TEMPLATE_RANK に設定される値である)。INTENT (OUT) 引数とする。UB の i 番目の要素には、テンプレートの i 番目の次元軸に対する *align-target* の宣言時の上限が設定される。

[3] 訳注: 原文の誤り。HPF 2.0 では「結合被覆」の定義は抹消されている。ALIGNEE は結合変数グループのメンバであってはならないと解釈すべきである。

AXIS_TYPE (省略可能) 基本文字型の 1 次元配列でなければならない。長さは任意であるが、完全な値を得るためには少なくとも長さが 10 必要である。各要素には以下の値が文字組込み代入文と同じ方法で設定される。少なくとも ALIGNEE が最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは TEMPLATE_RANK に設定される値である)。INTENT (OUT) 引数とする。AXIS_TYPE の i 番目の要素には、*align-target* の i 番目の次元軸に関する情報が設定される。HPF では以下の値が定義されている (実装によりその他の値が定義されていることがある)。

'NORMAL' *align-target* のこの次元軸には、ALIGNEE のある次元軸が整列している。この値が代入された AXIS_TYPE の要素に対しては、対応する AXIS_INFO の要素に、この *align-target* の次元軸に整列する ALIGNEE の次元軸の番号が設定される。

'REPLICATED' *align-target* のこの次元軸には、ALIGNEE が複製されている。この値が代入された AXIS_TYPE の要素に対しては、対応する AXIS_INFO の要素に、この *align-target* の次元軸に複製された ALIGNEE の複製の数が設定される。

'SINGLE' *align-target* のこの次元軸のある座標に、ALIGNEE が整列している。この値が代入された AXIS_TYPE の要素に対しては、対応する AXIS_INFO の要素に、ALIGNEE が整列している *align-target* の座標が設定される。

AXIS_INFO (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE が最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは TEMPLATE_RANK に設定される値である)。INTENT (OUT) 引数とする。上記 AXIS_TYPE の項を参照されたい。

NUMBER_ALIGNED (省略可能) 基本整数型スカラでなければならない。INTENT (OUT) 引数とする。この最終 *align-target* に整列している変数の総数が設定される。これは、この *align-target* が再分散されたときに同時に移動する変数の数である。

例示. HPF_ALIGNMENT の例に示す宣言が与えられ、実際のマッピングが宣言通りに行なわれたとすると、HPF_TEMPLATE の呼出しの結果は以下のようになる。

	A	C	D
LB	[1, 1]	[1, 1]	[1, 1]
UB	[40, 20]	[40, 20]	[40, 20]
AXIS_TYPE	['NORMAL', 'NORMAL']	['NORMAL', 'NORMAL']	['NORMAL', 'SINGLE']
AXIS_INFO	[1, 2]	[3, 1]	[1, 4]
NUMBER_ALIGNED	3	3	3
TEMPLATE_RANK	2	2	2

IALL(ARRAY, DIM, MASK)

省略可能な引数. DIM, MASK

機能. DIM 番目の次元について、ARRAY の要素のビット論理積を集計する。

分類. 変型関数

引数.

ARRAY　整数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。対応する実引数は、省略可能な仮引数であってはならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. 結果は、ARRAY と同じ種別型パラメタの整数型とする。DIMが省略されているか、又はARRAYが 1 次元配列であるときは、スカラとする。それ以外のとき、結果は $n-1$ 次元配列とし、形状は $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ とする。ここで $(d_1, d_2, \ldots, d_n)$ は、ARRAY の形状とする。

結果の値.

場合 *(i)*: IALL(ARRAY) の結果の値は、ARRAY の全要素の IAND 集計とする。ただし、ARRAY の大きさが 0 であるときは、ARRAY と同じ型種別パラメタのすべての整数 I に対して IAND(I, x) = I なる性質を持つ実装依存の値 x とする。7.4.3 項を参照のこと。

場合 *(ii)*: IALL(ARRAY, MASK=MASK) の結果の値は、MASK 中の真である要素と対応している ARRAY の要素の IAND 集計とする。ただし、真である要素がないときは、ARRAY と同じ型種別パラメタのすべての整数 I に対して IAND(I, x) = I なる性質を持つ実装依存の値 x とする。

場合 *(iii)*: ARRAY が 1 次元配列のとき、IALL(ARRAY, DIM [,MASK]) の結果の値は IALL(ARRAY [,MASK]) とする。2 次元以上のとき、IALL(ARRAY, DIM [,MASK]) の要素 $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ の値は 、IALL(ARRAY($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$) [,MASK = MASK($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$)]) とする。

例示.

場合 *(i)*: IALL((/ 7, 6, 3, 2 /)) の値は、2 になる。

場合 *(ii)*: IALL(C, MASK = BTEST(C,0)) の値は、C の奇数である要素の IAND 集計になる。

場合 *(iii)*: 配列 B が $\begin{bmatrix} 2 & 3 & 5 \\ 3 & 7 & 7 \end{bmatrix}$ であるとき、IALL(B, DIM = 1) の値は $\begin{bmatrix} 2 & 3 & 5 \end{bmatrix}$ になり、IALL(B, DIM = 2) の値は $\begin{bmatrix} 0 & 3 \end{bmatrix}$ になる。

IALL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに順方向のビット論理積累計を計算する。

分類. 変型関数

引数.

ARRAY　整数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、IALL((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示. IALL_PREFIX((/1,3,2,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 1 & 1 & 0 & 4 & 4 \end{bmatrix}$ となる。

IALL_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

省略可能な引数. MASK

機能. MASK で選択される ARRAY の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の要素の第 j ビットは、BASE の対応する要素及びその位置に拡散される ARRAY の要素の第 j ビットがすべて 1 であるとき、かつこのときに限り、1 となる。

分類. 変型関数

引数.

ARRAY　整数型でなければならず、スカラであってはならない。

BASE　ARRAY と同じ種別型パラメタの整数型でなければならず、スカラであってはならない。

INDX1,...,INDXn 整数型でなければならず、ARRAY と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、IALL((/$a_1, a_2, ..., a_m, b$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる ARRAY の要素とする。

例示. IALL_SCATTER((/ 1, 2, 3, 6 /), (/ 1, 3, 7 /), (/ 1, 1, 2, 2 /)) は $\begin{bmatrix} 0 & 2 & 7 \end{bmatrix}$ となる。

IALL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに逆方向のビット論理積累計を計算する。

分類. 変型関数

引数.

ARRAY 整数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、IALL((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示. IALL_SUFFIX((/1,3,2,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 0 & 2 & 2 & 4 & 5 \end{bmatrix}$ となる。

IANY(ARRAY, DIM, MASK)

省略可能な引数. DIM, MASK

機能. DIM 番目の次元について、ARRAY の要素のビット論理和を集計する。

分類. 変型関数

引数.

ARRAY　整数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。対応する実引数は、省略可能な仮引数であってはならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. 結果は、ARRAY と同じ種別型パラメタの整数型とする。DIMが省略されているか、又はARRAYが 1 次元配列であるときは、スカラとする。それ以外のとき、結果は $n-1$ 次元配列とし、形状は $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ とする。ここで $(d_1, d_2, \ldots, d_n)$ は、ARRAY の形状とする。

結果の値.

場合 *(i)*: IANY(ARRAY) の結果の値は、ARRAY の全要素の IOR 集計とする。ただし、ARRAY の大きさが 0 であるときは、0 とする。7.4.3 項を参照のこと。

場合 *(ii)*: IANY(ARRAY, MASK=MASK) の結果の値は、MASK 中の真である要素と対応している ARRAY の要素の IOR 集計とする。ただし、真である要素がないときは、0 とする。

場合 *(iii)*: ARRAY が 1 次元配列のとき、IANY(ARRAY, DIM [,MASK]) の結果の値は IANY(ARRAY [,MASK]) とする。2 次元以上のとき、IANY(ARRAY, DIM [,MASK]) の要素 $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ の値は 、IANY(ARRAY($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$) [,MASK = MASK($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$)]) とする。

例示.

場合 *(i)*: IANY((/ 9, 8, 3, 2 /)) の値は、11 になる。

場合 *(ii)*: IANY(C, MASK = BTEST(C,0)) の値は、C の奇数である要素の IOR 集計になる。

場合 *(iii)*: 配列 B が $\begin{bmatrix} 2 & 3 & 5 \\ 0 & 4 & 2 \end{bmatrix}$ であるとき、IANY(B, DIM = 1) の値は $\begin{bmatrix} 2 & 7 & 7 \end{bmatrix}$ になり、IANY(B, DIM = 2) の値は $\begin{bmatrix} 7 & 6 \end{bmatrix}$ になる。

IANY_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに順方向のビット論理和累計を計算する。

分類. 変型関数

引数.

ARRAY　整数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、IANY((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示. IANY_PREFIX((/1,2,3,2,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 1 & 3 & 3 & 2 & 7 \end{bmatrix}$ となる。

IANY_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

省略可能な引数. MASK

機能. MASK で選択される ARRAY の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の要素の第 j ビットは、BASE の対応する要素またはその位置に拡散される ARRAY の要素の第 j ビットの中に一つでも 1 があるとき、かつこのときに限り、1 となる。

分類. 変型関数

引数.

ARRAY　整数型でなければならず、スカラであってはならない。

BASE　ARRAY と同じ種別型パラメタの整数型でなければならず、スカラであってはならない。

INDX1,...,INDXn 整数型でなければならず、ARRAY と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、IANY((/$a_1, a_2, ..., a_m, b$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる ARRAY の要素とする。

例示. IANY_SCATTER((/ 1, 2, 3, 6 /), (/ 1, 3, 7 /), (/ 1, 1, 2, 2 /)) は $\begin{bmatrix} 3 & 7 & 7 \end{bmatrix}$ となる。

IANY_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに逆方向のビット論理和累計を計算する。

分類. 変型関数

引数.

ARRAY　整数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、IANY((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示. IANY_SUFFIX((/4,2,3,2,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 7 & 3 & 3 & 7 & 5 \end{bmatrix}$ となる。

IPARITY(ARRAY, DIM, MASK)

省略可能な引数. DIM, MASK

機能. DIM 番目の次元について、ARRAY の要素のビット排他的論理和を集計する。

分類. 変型関数

引数.

ARRAY　整数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。対応する実引数は、省略可能な仮引数であってはならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. 結果は、ARRAY と同じ種別型パラメタの整数型とする。DIM が省略されているか、又は ARRAY が 1 次元配列であるときは、スカラとする。それ以外のとき、結果は $n-1$ 次元配列とし、形状は $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ とする。ここで $(d_1, d_2, \ldots, d_n)$ は、ARRAY の形状とする。

結果の値.

場合 *(i)*: IPARITY(ARRAY) の結果の値は、ARRAY の全要素の IEOR 集計とする。ただし、ARRAY の大きさが 0 であるときは、0 とする。7.4.3 項を参照のこと。

場合 *(ii)*: IPARITY(ARRAY, MASK=MASK) の結果の値は、MASK 中の真である要素と対応している ARRAY の要素の IEOR 集計とする。ただし、真である要素がないときは、0 とする。

場合 *(iii)*: ARRAY が 1 次元配列のとき、IPARITY(ARRAY, DIM [,MASK]) の結果の値は IPARITY(ARRAY [,MASK]) とする。2 次元以上のとき、IPARITY(ARRAY, DIM [,MASK]) の要素 $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ の値は、IPARITY(ARRAY($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$) [,MASK = MASK($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$)]) とする。

例示.

場合 *(i)*: IPARITY((/ 13, 8, 3, 2 /)) の値は、4 になる。

場合 *(ii)*: IPARITY(C, MASK = BTEST(C,0)) の値は、C の奇数である要素の IEOR 集計になる。

場合 *(iii)*: 配列 B が $\begin{bmatrix} 2 & 3 & 7 \\ 0 & 4 & 2 \end{bmatrix}$ であるとき、IPARITY(B, DIM = 1) の値は $\begin{bmatrix} 2 & 7 & 5 \end{bmatrix}$ になり、IPARITY(B, DIM = 2) の値は $\begin{bmatrix} 6 & 6 \end{bmatrix}$ になる。

IPARITY_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに順方向のビット排他的論理和累計を計算する。

分類. 変型関数

引数.

ARRAY 整数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、IPARITY((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示. IPARITY_PREFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 1 & 3 & 0 & 4 & 1 \end{bmatrix}$ となる。

IPARITY_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

省略可能な引数. MASK

機能. MASK で選択される ARRAY の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の要素の第 j ビットは、BASE の対応する要素とその位置に拡散される ARRAY の要素の第 j ビットの中に奇数個の 1 があるとき、かつこのときに限り、1 となる。

分類. 変型関数

引数.

ARRAY　整数型でなければならず、スカラであってはならない。

BASE　ARRAY と同じ 種別型パラメタの整数型でなければならず、スカラであってはならない。

INDX1,...,INDXn 整数型でなければならず、ARRAY と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、IPARITY((/$a_1, a_2, ..., a_m, b$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる ARRAY の要素とする。

例示. IPARITY_SCATTER((/ 1, 2, 3, 6 /), (/ 1, 3, 7 /), (/ 1, 1, 2, 2 /)) は $\begin{bmatrix} 2 & 6 & 7 \end{bmatrix}$ となる。

IPARITY_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに逆方向のビット排他的論理和累計を計算する。

分類. 変型関数

引数.

ARRAY　　整数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値.　結果の要素 r の値は、IPARITY((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示.　IPARITY_SUFFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 0 & 1 & 3 & 1 & 5 \end{bmatrix}$ となる。

LEADZ(I)

機能. 整数の先頭の 0 であるビットの個数を返す。

分類. 要素別処理関数

引数. I は整数型でなければならない。

結果の型 及び 型パラメタ. I と同じとする。

結果の値. 結果は整数 I の先頭の 0 であるビットの個数とする。整数値をビット列として解釈する数体系は、F95:13.5.7 項による。LEADZ(0) の値は、BIT_SIZE(I) とする。I が 0 でないとき、立っている最も左のビットが第 $k-1$ ビットであれば (ただし右端を第 0 ビットとする)、LEADZ(I) の結果は BIT_SIZE(I) $-\ k$ とする。

例示.　LEADZ(3) の値は BIT_SIZE(3) - 2 になる。スカラ I に対し、LEADZ(I) .EQ. MINVAL((/ (J, J=0, BIT_SIZE(I)) /), MASK=M) になる。ここで、M =(/ (BTEST(I,J), J=BIT_SIZE(I)-1, 0, -1), .TRUE. /) とする。ある整数 I について、LEADZ(I) の結果は、整数の表現形式のビット数 (BIT_SIZE(I)) により異なることがある。これは、LEADZ がビット数を最上位ビットから数えるためである。ILEN と比較されたい。

MAXVAL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに順方向の MAXVAL 累計を計算する。

分類. 変型関数

引数.

ARRAY　整数型又は実数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq \mathtt{DIM} \leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値.　結果の要素 r の値は、MAXVAL((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示.　MAXVAL_PREFIX((/3,4,-5,2,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 3 & 4 & 4 & 2 & 5 \end{bmatrix}$ となる。

MAXVAL_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

省略可能な引数. MASK

機能. MASK で選択される ARRAY の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の各要素は、BASE の対応する要素とその位置に拡散される ARRAY の要素の最大値となる。

分類. 変型関数

引数.

ARRAY　整数型又は実数型でなければならず、スカラであってはならない。

BASE　ARRAY と同じ型及び同じ種別型パラメタでなければならず、スカラであってはならない。

INDX1,...,INDXn 整数型でなければならず、ARRAY と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、MAXVAL((/$a_1, a_2, ..., a_m, b$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる ARRAY の要素とする。

例示. MAXVAL_SCATTER((/ 1, 2, 3, 1 /), (/ 4, -5, 7 /), (/ 1, 1, 2, 2 /)) は $\begin{bmatrix} 4 & 3 & 7 \end{bmatrix}$ となる。

MAXVAL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに逆方向の MAXVAL 累計を計算する。

分類. 変型関数

引数.

ARRAY　整数型又は実数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、MAXVAL((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示. MAXVAL_SUFFIX((/3,4,-5,2,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 4 & 4 & -5 & 5 & 5 \end{bmatrix}$ となる。

MINVAL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに順方向の MINVAL 累計を計算する。

分類. 変型関数

引数.

ARRAY　整数型又は実数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、MINVAL((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示. MINVAL_PREFIX((/1,2,-3,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 1 & 1 & -3 & 4 & 4 \end{bmatrix}$ となる。

MINVAL_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

省略可能な引数. MASK

機能. MASK で選択される ARRAY の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の各要素は、BASE の対応する要素とその位置に拡散される ARRAY の要素の最小値となる。

分類. 変型関数

引数.

ARRAY　整数型又は実数型でなければならず、スカラであってはならない。

BASE　ARRAY と同じ型及び同じ種別型パラメタでなければならず、スカラであってはならない。

INDX1,...,INDXn 整数型でなければならず、ARRAY と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、MINVAL((/$a_1, a_2, ..., a_m, b$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる ARRAY の要素とする。

例示. MINVAL_SCATTER((/ 1, -2, -3, 6 /), (/ 4, 3, 7 /), (/ 1, 1, 2, 2 /)) は $\begin{bmatrix} -2 & -3 & 7 \end{bmatrix}$ となる。

MINVAL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに逆方向の MINVAL 累計を計算する。

分類. 変型関数

引数.

ARRAY　　整数型又は実数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値.　結果の要素 r の値は、MINVAL((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示.　MINVAL_SUFFIX((/1,2,-3,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} -3 & -3 & -3 & 4 & 5 \end{bmatrix}$ となる。

PARITY(MASK, DIM)

省略可能な引数. DIM

機能. MASK の第 DIM 次元の要素の値に奇数個の真があるかどうか判定する。

分類. 変型関数

引数.

MASK　　論理型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、MASK の次元数とする。対応する実引数は、省略可能な仮引数であってはならない。

結果の型、型パラメタ 及び 形状. 結果は、MASK と同じ種別型パラメタの論理型とする。DIM が省略されているか、又は MASK が 1 次元配列であるときは、スカラとする。それ以外のとき、結果は $n-1$ 次元配列とし、形状は $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ とする。ここで $(d_1, d_2, \ldots, d_n)$ は、MASK の形状とする。

結果の値.

場合 *(i):* PARITY(MASK) の結果の値は、MASK の全要素の .NEQV. 集計とする。ただし、MASK の大きさが 0 であるときは、偽とする。7.4.3 項を参照のこと。

場合 *(ii):* MASK が 1 次元配列のとき、PARITY(MASK, DIM) の結果の値は PARITY(MASK) とする。2 次元以上のとき、PARITY(MASK, DIM) の要素 $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ の値は、PARITY(MASK($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$)) とする。

例示.

場合 *(i):* PARITY((/ T, T, T, F /)) の値は、真になる。

場合 *(ii):* 配列 B が $\begin{bmatrix} \mathrm{T} & \mathrm{T} & \mathrm{F} \\ \mathrm{T} & \mathrm{T} & \mathrm{T} \end{bmatrix}$ であるとき、PARITY(B, DIM = 1) の値は $\begin{bmatrix} \mathrm{F} & \mathrm{F} & \mathrm{T} \end{bmatrix}$ になり、PARITY(B, DIM = 2) の値は $\begin{bmatrix} \mathrm{F} & \mathrm{T} \end{bmatrix}$ になる。

PARITY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, SEGMENT, EXCLUSIVE

機能. MASK の DIM 番目の次元について、区分ごとに順方向の排他的論理和累計を計算する。

分類. 変型関数

引数.

MASK 論理型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、MASK の次元数とする。

SEGMENT (省略可能) 論理型でなければならず、MASK と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. MASK と同じとする。

結果の値. 結果の要素 r の値は、PARITY((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された MASK の要素の集合である (空集合のこともある)。

例示. PARITY_PREFIX((/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{T} & \mathrm{F} \end{bmatrix}$ となる。

PARITY_SCATTER(MASK,BASE,INDX1, ..., INDXn)

機能. MASK の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の要素は、BASE の対応する要素とその位置に拡散される MASK の要素の中に奇数個の真があるとき、かつそのときに限り、真となる。

分類. 変型関数

引数.

MASK　論理型でなければならず、スカラであってはならない。

BASE　MASK と同じ種別型パラメタの論理型でなければならず、スカラであってはならない。

INDX1,...,INDXn 整数型でなければならず、MASK と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、PARITY((/$a_1, a_2, ..., a_m, b$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる MASK の要素とする。

例示. PARITY_SCATTER((/ T, T, T, T /), (/ T, F, F /), (/ 1, 1, 1, 2 /)) は $\begin{bmatrix} \texttt{F} & \texttt{T} & \texttt{F} \end{bmatrix}$ となる。

PARITY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, SEGMENT, EXCLUSIVE

機能. MASK の DIM 番目の次元について、区分ごとに逆方向の排他的論理和累計を計算する。

分類. 変型関数

引数.

MASK　論理型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、MASK の次元数とする。

SEGMENT (省略可能) 論理型でなければならず、MASK と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. MASK と同じとする。

結果の値. 結果の要素 r の値は、PARITY((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された MASK の要素の集合である (空集合のこともある)。

例示. PARITY_SUFFIX((/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} \mathrm{F} & \mathrm{T} & \mathrm{T} & \mathrm{F} & \mathrm{T} \end{bmatrix}$ となる。

POPCNT(I)

機能. 整数中の 1 であるビットの個数を返す。

分類. 要素別処理関数

引数. I は整数型でなければならない。

結果の型 及び 型パラメタ. I と同じとする。

結果の値. POPCNT(I) は、整数 I の 2 進表現中の 1 であるビットの個数とする。整数値をビット列として解釈する数体系は、F95:13.5.7 項による。

例示. スカラ I に対し、POPCNT(I) = COUNT((/ (BTEST(I,J), J=0, BIT_SIZE(I)-1) /)) になる。

POPPAR(I)

機能. 整数中のビットの奇偶を返す。

分類. 要素別処理関数

引数. I は整数型でなければならない。

結果の型 及び 型パラメタ. I と同じとする。

結果の値. POPPAR(I) は、整数 I 中に 1 であるビットが奇数個あるとき 1 とし、偶数個あるとき 0 とする。整数値をビット列として解釈する数体系は、F95:13.5.7 項による。

例示. スカラ I に対し、POPPAR(I) = MERGE(1,0,BTEST(POPCNT(I),0)) になる。

PRODUCT_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに順方向の積の累計を計算する。

分類. 変型関数

引数.

ARRAY　整数型、実数型又は複素数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値.　結果の要素 r の値は、PRODUCT((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示.　PRODUCT_PREFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 1 & 2 & 6 & 4 & 20 \end{bmatrix}$ となる。

PRODUCT_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

省略可能な引数. MASK

機能. MASK で選択される ARRAY の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の各要素は、BASE の対応する要素とその位置に拡散される ARRAY の要素の積となる。

分類. 変型関数

引数.

ARRAY　整数型、実数型又は複素数型でなければならず、スカラであってはならない。

BASE　ARRAY と同じ型及び同じ種別型パラメタでなければならず、スカラであってはならない。

INDX1,...,INDXn 整数型でなければならず、ARRAY と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、PRODUCT((/$a_1, a_2, ..., a_m, b$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる ARRAY の要素とする。

例示.　PRODUCT_SCATTER((/ 1, 2, 3, 1 /), (/ 4, -5, 7 /), (/ 1, 1, 2, 2 /)) は $\begin{bmatrix} 8 & -15 & 7 \end{bmatrix}$ となる。

PRODUCT_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに逆方向の積の累計を計算する。

分類. 変型関数

引数.

ARRAY　　整数型、実数型又は複素数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値.　結果の要素 r の値は、PRODUCT((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示.　PRODUCT_SUFFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 6 & 6 & 3 & 20 & 5 \end{bmatrix}$ となる。

SORT_DOWN(ARRAY,DIM)

省略可能な引数. DIM

機能. 降順にソートする。

分類. 変型関数

引数.

ARRAY　　整数型、実数型又は文字型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。対応する実引数は、省略可能な仮引数であってはならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値.

場合 *(i)*: ARRAY が 1 次元配列のとき、SORT_DOWN(ARRAY) の結果は、ARRAY と同じ形状のベクトルで、ARRAY と同じ要素を (同じ出現数だけ) 含み、要素が降順にソートされたものとする。文字型配列に対する大小順序は、Fortran の組込み関数で使われるもの、すなわち ASCII 大小順序とする。

場合 *(ii)*: ARRAY が多次元配列のとき、SORT_DOWN(ARRAY) の結果は、ARRAY を配列要素順序で 1 次元配列 V に変形し、場合 1 の方法でこの 1 次元配列を降順にソートし、最後にそれを元の形状に戻して得られる結果とする。すなわち、RESHAPE(SORT_DOWN(V), SHAPE = SHAPE(ARRAY)) と同じ結果とする。ここで、V = RESHAPE(ARRAY, SHAPE = (/ M /) とし、M = SIZE(ARRAY) とする。

場合 *(iii)*: SORT_DOWN(ARRAY, DIM=k) の結果は、ARRAY と同じ要素を含むが、各々の 1 次元部分配列 ARRAY$(i_1, i_2, \ldots, i_{k-1}, :, i_{k+1}, \ldots, i_n)$ が Case(i) の方法で要素の降順にソートされたものとする。ここで、n は ARRAY の次元数とする。

例示.

場合 *(i)*: SORT_DOWN((/30, 20, 30, 40, -10/)) の値は $\begin{bmatrix} 40 & 30 & 30 & 20 & -10 \end{bmatrix}$ になる。

場合 *(ii)*: 配列 A が $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$ であるとき、SORT_DOWN(A) の値は $\begin{bmatrix} 9 & 4 & 2 \\ 5 & 2 & 1 \\ 4 & 2 & 1 \end{bmatrix}$ になる。

場合 *(iii)*: 配列 A が $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$

であるとき、SORT_DOWN(A, DIM = 1) の値は $\begin{bmatrix} 4 & 9 & 4 \\ 1 & 5 & 2 \\ 1 & 2 & 2 \end{bmatrix}$ になる。

SORT_UP(ARRAY,DIM)

省略可能な引数. DIM

機能. 昇順にソートする。

分類. 変型関数

引数.

ARRAY 整数型、実数型又は文字型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。対応する実引数は、省略可能な仮引数であってはならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値.

場合 *(i):* ARRAY が 1 次元配列のとき、SORT_UP(ARRAY) の結果は、ARRAY と同じ形状のベクトルで、ARRAY と同じ要素を (同じ出現数だけ) 含み、要素が昇順にソートされたものとする。文字型配列に対する大小順序は、Fortran の組込み関数で使われるもの、すなわち ASCII 大小順序とする。

場合 *(ii):* ARRAY が多次元配列のとき、SORT_UP(ARRAY) の結果は、ARRAY を配列要素順序で 1 次元配列 V に変形し、場合 1 の方法でこの 1 次元配列を昇順にソートし、最後にそれを元の形状に戻して得られる結果とする。すなわち、RESHAPE(SORT_UP(V), SHAPE = SHAPE(ARRAY)) と同じ結果とする。ここで、V = RESHAPE(ARRAY, SHAPE = (/ M /) とし、M = SIZE(ARRAY) とする。

場合 *(iii):* SORT_UP(ARRAY, DIM=k) の結果は、ARRAY と同じ要素を含むが、各々の 1 次元部分配列 ARRAY$(i_1, i_2, \ldots, i_{k-1}, :, i_{k+1}, \ldots, i_n)$ が Case(i) の方法で要素の昇順にソートされたものとする。ここで、n は ARRAY の次元数とする。

例示.

場合 *(i):* SORT_UP((/30, 20, 30, 40, -10/)) の値は $\begin{bmatrix} -10 & 20 & 30 & 30 & 40 \end{bmatrix}$ になる。

場合 *(ii):* 配列 A が $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$ であるとき、SORT_UP(A) の値は $\begin{bmatrix} 1 & 2 & 4 \\ 1 & 2 & 5 \\ 2 & 4 & 9 \end{bmatrix}$ になる。

場合 *(iii):* 配列 A が $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$ であるとき、SORT_UP(A, DIM = 1) の値は $\begin{bmatrix} 1 & 2 & 2 \\ 1 & 5 & 2 \\ 4 & 9 & 4 \end{bmatrix}$ になる。

SUM_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに順方向の SUM 累計を計算する。

分類. 変型関数

引数.

ARRAY　整数型、実数型又は複素数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、SUM((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示. SUM_PREFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 1 & 3 & 6 & 4 & 9 \end{bmatrix}$. となる。

SUM_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

省略可能な引数. MASK

機能. MASK で選択される ARRAY の要素を、指標配列 INDX1, ..., INDXn で指定される結果の位置に拡散する。結果の各要素は、BASE の対応する要素とその位置に拡散される ARRAY の要素の和となる。

分類. 変型関数

引数.

ARRAY 整数型、実数型又は複素数型でなければならず、スカラであってはならない。

BASE ARRAY と同じ型及び同じ種別型パラメタでなければならず、スカラであってはならない。

INDX1,...,INDXn 整数型でなければならず、ARRAY と形状適合しなければならない。引数 INDX の数は、BASE の次元数と等しくなければならない。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

結果の型、型パラメタ 及び 形状. BASE と同じとする。

結果の値. BASE の要素 b に対応する結果の要素の値は、SUM((/$a_1, a_2, ..., a_m, b$/)) とする。ここで $(a_1, \ldots, a_m)$ は 7.4.4 項に述べた方法で b と関連付けられる ARRAY の要素とする。

例示. SUM_SCATTER((/ 1, 2, 3, 1 /), (/ 4, -5, 7 /), (/ 1, 1, 2, 2 /)) は $\begin{bmatrix} 7 & -1 & 7 \end{bmatrix}$ となる。

SUM_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)

省略可能な引数. DIM, MASK, SEGMENT, EXCLUSIVE

機能. ARRAY の DIM 番目の次元について、区分ごとに逆方向の SUM 累計を計算する。

分類. 変型関数

引数.

ARRAY　　整数型、実数型又は複素数型でなければならず、スカラであってはならない。

DIM (省略可能) 整数型スカラであって、$1 \leq \mathtt{DIM} \leq n$ の範囲の値でなければならない。ここで n は、ARRAY の次元数とする。

MASK (省略可能) 論理型でなければならず、ARRAY と形状適合しなければならない。

SEGMENT (省略可能) 論理型でなければならず、ARRAY と同じ形状でなければならない。

EXCLUSIVE (省略可能) 論理型スカラでなければならない。

結果の型、型パラメタ 及び 形状. ARRAY と同じとする。

結果の値. 結果の要素 r の値は、SUM((/ $a_1, \ldots, a_m$ /)) とする。ここで、$(a_1, \ldots, a_m)$ は、7.4.5 項に述べた規則により、r に寄与すべく選択された ARRAY の要素の集合である (空集合のこともある)。

例示.　SUM_SUFFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) の値は $\begin{bmatrix} 6 & 5 & 3 & 9 & 5 \end{bmatrix}$ となる。

第III部

HPF公認拡張仕様

ここでは、High Performance Fortran 公認拡張の機能の構文と意味について記述する。ほとんどの場合、これらの機能は HPF の概念に基づいて設計されている。そのため、背景となる情報を知るためには第 I 部や第 II 部を参照しなければならない。

第8章　データマッピングの公認拡張仕様

本章では、第 3 章で述べたデータマッピングに関する基本機能の能力を拡張するいくつかの機能について述べる。これらの拡張は、二つのカテゴリーに分けられる。

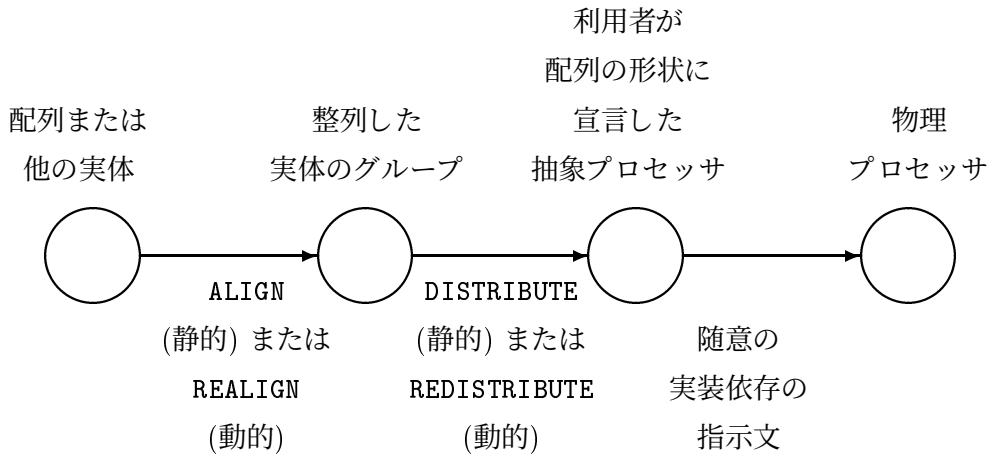
拡張の第一のカテゴリーでは、利用者がデータのマッピングに関してさらに多くの制御を行なうための機能を提供する。それらには、DYNAMIC と宣言したデータを実行時に再分散や再整列することができる動的再マッピングのための指示文が含まれる。DISTRIBUTE 指示文の ONTO 節は、プロセッサ集合の部分への分散を可能にするように拡張される。さらに、ポインタや構造型の成分の明示的マッピングも導入される。二つの新しい分散も含まれる: ブロック分散を一般化した GEN_BLOCK 分散、それに、個々の配列要素のマッピングをマッピング配列を使って指定することを可能にする INDIRECT 分散である。

拡張の第二のカテゴリーは、利用者が効率的なプログラムを作成するために、有益な情報をコンパイラに与えることに利用することができる。RANGE 指示文を使うと、動的に分散された配列、ポインタまたは仮引数の分散のレンジを指定することができる。SHADOW 指示文は、隣接計算 (nearest-neighbor computation) において、ローカルでない要素を収めるための追加スペースの量を指定するために利用することができる。

本章は、拡張について述べるため、第 3 章および第 4 章の一部を繰り返し述べる。その際、必要に応じて新しい構文規則を示し、古い構文規則を拡張する。特に、8.13、8.14 および 8.15 節は、ここで述べる公認拡張に基づいて第 3 章の対応する節を拡張する。

8.1　拡張されたモデル

抽象プロセッサへのデータの割当ての基本モデルは、第 3 章で述べたような二段階マッピングのままである。しかし、以下に図示されているように、データ実体の動的再マッピングを可能にするように拡張されている:



したがって、実行指示文 REALIGN および REDISTRIBUTE を使って実体を実行時に再マップすることができる。整列木の根である実体 (すなわち、他の実体に明示的に整列していない実体) は、明示的に再分散することができる。そのような実体を再分散することは、整列関係を保つために、その実体に最終的に整列しているすべての実体の再分散を引き起こす。

整列木の根でない実体は、明示的に再整列することができるが、明示的に再分散することはできない。このような再整列は、他の実体のマッピングを変更することはない。データの再マッピングは、プロセッサ間の通信を必要とすることに注意されたい。

HPF は、Fortran の ALLOCATABLE 属性への類推によって、DYNAMIC 属性を採り入れた。DYNAMIC と宣言されていない配列は、REALIGN することが許されない。同様に、DYNAMIC と宣言されていない配列やテンプレートは、REDISTRIBUTE することが許されない。

SAVE 属性付きの局所変数、共通ブロックの変数および参照結合により参照される変数は、暗黙的に再マップ (たとえば、可変的な分散形式をもつことによって、または可変的な分散形式をもつ言語要素に整列することによって起きる) されてはならない。これらの三つの種類の変数の中では、参照結合によりアクセスされる変数だけが DYNAMIC 属性をもつことができる。

3.1 節に示すように、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。3.1 節のマッピング指示文の定義は、次のように拡張される: マッピング指示文とは、ALIGN、DISTRIBUTE、INHERIT、DYNAMIC、RANGE、SHADOW 指示文、それに、整列や分散または INHERIT、DYNAMIC、RANGE、SHADOW 属性を与える指示文である。

8.2 拡張データマッピング指示文の属性形式の構文

他のマッピング指示文と同様に、実行指示文である REALIGN や REDISTRIBUTE にも二つの形式 (文形式と属性形式) がある。しかし、一つの指示文の中で他の属性と組み合わせることはできない。RANGE と SHADOW 属性は、一つの指示文の中で他の属性と組み合わせることができる。

H801 *combined-attribute-extended* **is** ALIGN *align-attribute-stuff*
or DISTRIBUTE *dist-attribute-stuff*
or INHERIT
or TEMPLATE
or PROCESSORS
or DIMENSION (*explicit-shape-spec-list*)
or DYNAMIC
or RANGE *range-attr-stuff*
or SHADOW *shadow-attr-stuff*
or SUBSET

制約: SUBSET 属性は、プロセッサ構成にだけ適用できる。

SUBSET 属性については、第 9 章で論じる。他は下記で論じる。

8.3 REDISTRIBUTE 指示文

REDISTRIBUTE 指示文は、DISTRIBUTE 指示文と似ている。ただし、REDISTRIBUTE 指示文は実行指示文である。実体やテンプレートは、DYNAMIC と宣言されていればいつでも再分散することができる (8.5 節 参照)。配列 (またはテンプレート) に対し最終的に整列している他の実体は、その最終整列先が再分散されたときに、新しい分散を反映するために再マップされ、整列関係を維持する (3.4 節 参照)。(これらの操作は、実行時に多くの計算と通信の労力を必要とするため、プログラマは、この機能を使う場合には注意する必要がある。)

DISTRIBUTE 指示文は、有効域の宣言部にだけ現れることができる。REDISTRIBUTE 指示文は、有効域の実行部にだけ現れることができる。DISTRIBUTE と REDISTRIBUTE の主要な違いは、DISTRIBUTE は、分散形式 (BLOCK や CYCLIC など) の引数として、宣言式 (*specification-expr*) だけを含むことができ、一方 REDISTRIBUTE では、これらの引数はどんな整数式でも構わないことである。もう一つの違いは、DISTRIBUTE は属性であり、*combined-directive* の一部として他の属性と組み合わせることができるが、REDISTRIBUTE は属性でない。(ただし、REDISTRIBUTE 文は、"::" 区切りを使った属性形式で記述することができる。)

REDISTRIBUTE 指示文の構文は次のとおりである。

H802 *redistribute-directive* **is** REDISTRIBUTE *distributee dist-directive-stuff*
or REDISTRIBUTE *dist-attribute-stuff* :: *distributee-list*

制約: REDISTRIBUTE 指示文に現れる *distributee* は、DYNAMIC 属性をもたなければならない (8.5 節 参照)。

制約: REDISTRIBUTE 指示文の *distributee* は、ALIGN または REALIGN 指示文の *alignee* として現れてはならない。

制約: REDISTRIBUTE 指示文の *dist-format-clause* や *dist-target* のどちらも、"*" で始まってはならない。

実体は、INHERIT 属性と DISTRIBUTE 属性の両方をもってはならないものの、INHERIT 属性をもっているいないにかかわらず、DYNAMIC 属性をもち、ALIGN や REALIGN 指示文の *alignee* に現れない限り、REDISTRIBUTE 指示文の *distributee* に現れることができることに注意されたい。

RANGE 指示文 (8.11 節 参照) が、*distributee* で許される分散形式を限定するために使われた場合、新しいマッピングは、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。

REDISTRIBUTE 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$ REDISTRIBUTE distributee ( dist-format-list ) ONTO dist-target
```

は、以下と同じである。

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

8.4 REALIGN 指示文

REALIGN 指示文は、ALIGN 指示文と似ている。ただし、REALIGN 指示文は実行指示文である。配列 (またはテンプレート) は、DYNAMIC と宣言されていればいつでも再整列することができる (8.5 節 参照)。再分散 (8.5 節 参照) と違い、データ実体の再整列は、他の実体の再マッピングを引き起こさない。(しかし、たった一つの実体の再整列でも、それが大きければ実行時に多くの計算と通信の労力を必要とするため、プログラマはこの機能を使う場合には注意する必要がある。)

ALIGN 指示文は、有効域の宣言部にだけ現れることができる。REALIGN 指示文は、有効域の実行部にだけ現れることができる。ALIGN と REALIGN の主要な違いは、ALIGN は、添字 (*subscript*) や添字三つ組 (*subscript-triplet*) の中に宣言式 (*specification-expr*) だけを含むことができ、一方 REALIGN では、添字 (*subscript*) や添字三つ組 (*subscript-triplet*) の中の式は、宣言式 (*specification-expr*) である必要はないことである。もう一つの違いは、ALIGN は属性であり、*combined-directive* の一部として他の属性と組み合わせることができるが、REALIGN は属性でない。(ただし、REALIGN 文は、"::" 区切りを使った属性形式で記述することができる。)

REALIGN 指示文の構文は次のとおりである。

H803 *realign-directive* **is** REALIGN *alignee align-directive-stuff*
or REALIGN *align-attribute-stuff* :: *alignee-list*

制約: REALIGN 指示文に現れる *alignee* は、DYNAMIC 属性をもたなければならない (8.5 節 参照)。

制約: *align-with-clause* に指定された *align-target* が DYNAMIC 属性をもつ場合、*alignee* も同様に DYNAMIC 属性をもたなければならない。

制約: REALIGN 指示文の *alignee* は、DISTRIBUTE や REDISTRIBUTE 指示文の *distributee* であってはならない。

実体は、INHERIT 属性と ALIGN 属性の両方をもってはならないものの、INHERIT 属性をもっているいないにかかわらず、DYNAMIC 属性をもち、DISTRIBUTE や REDISTRIBUTE 指示文の *distributee* に現れない限り、REALIGN 指示文の *alignee* に現れることができることに注意されたい。

RANGE 指示文 (8.11 節 参照) が、*alignee* で許される分散形式を限定するために使われた場合、新しいマッピングは、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。

8.5 DYNAMIC 指示文

DYNAMIC 属性は、実体を動的に再整列や再分散してもよいことを指定する。

H804 *dynamic-directive* **is** DYNAMIC *alignee-or-distributee-list*

H805 *alignee-or-distributee* **is** *alignee*
or *distributee*

制約: 共通ブロックの実体は、DYNAMIC と宣言することはできず、また、DYNAMIC である実体 (またはテンプレート) に整列することはできない。(ここで制約されているようなことをしたいならば、共通ブロックの代わりにモジュールを使わなければならない。)

制約: 構造型の成分は、POINTER 属性をもつ場合だけ DYNAMIC 属性をもつことができる。(詳しくは、8.9 節を参照されたい。)

制約: SAVE 属性をもつ実体は、DYNAMIC と宣言することはできず、また、DYNAMIC である実体 (またはテンプレート) に整列することはできない。

REALIGN 指示文は、DYNAMIC 属性をもたない *alignee* に適用することはできない。REDISTRIBUTE 指示文は、DYNAMIC 属性をもたない *distributee* に適用することはできない。

DYNAMIC 指示文は、Fortran の属性に関する文法がそうであるように、他の指示文や任意の順序で並べた属性と組み合わせることができる。

例:

```
!HPF$ DYNAMIC A,B,C,D,E
!HPF$ DYNAMIC:: A,B,C,D,E
!HPF$ DYNAMIC, ALIGN WITH SNEEZY:: X,Y,Z
!HPF$ ALIGN WITH SNEEZY, DYNAMIC:: X,Y,Z
!HPF$ DYNAMIC, DISTRIBUTE(BLOCK, BLOCK) :: X,Y
!HPF$ DISTRIBUTE(BLOCK, BLOCK), DYNAMIC :: X,Y
```

最初の二つの例は、両方とも全く同じことを意味する。その次の二つの例は、両方とも全く同じことを意味する。最後の二つの例は、両方とも全く同じことを意味する。

以下の三つの指示文は、

```
!HPF$ TEMPLATE A(64,64),B(64,64),C(64,64),D(64,64)
!HPF$ DISTRIBUTE(BLOCK, BLOCK) ONTO P:: A,B,C,D
!HPF$ DYNAMIC A,B,C,D
```

次のように一つの指示文に組み合わせることができる。

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) ONTO P,   &
!HPF$   DIMENSION(64,64),DYNAMIC :: A,B,C,D
```

割付け実体も、DYNAMIC 属性をもってもよい。ALLOCATE 文の直後に REDISTRIBUTE や REALIGN 指示文が続く場合、その文法上の意味は、もし静的に宣言されたマッピングがあるならば、まずそれを使って配列が生成され、その後すぐに配列は再マップされる。しかし実際には、次のような明らかな最適化が可能である。すなわち、一回のステップで、配列を再マップされようとしているプロセッサで生成することである。HPF 実装者には、この最適化を実装することを勧告し、HPF プログラマには、この最適化に依存することを奨励する。ここで例を示す:

```
      REAL,ALLOCATABLE(:,:) :: TINKER, EVERS
!HPF$ DYNAMIC :: TINKER, EVERS
```

```
      REAL, ALLOCATABLE :: CHANCE(:)
!HPF$ DISTRIBUTE(BLOCK),DYNAMIC :: CHANCE
      ...
      READ 6,M,N
      ALLOCATE(TINKER(N*M,N*M))
!HPF$ REDISTRIBUTE TINKER(CYCLIC, BLOCK)
      ALLOCATE(EVERS(N,N))
!HPF$ REALIGN EVERS(:,:) WITH TINKER(M::M,1::M)
      ALLOCATE(CHANCE(10000))
!HPF$ REDISTRIBUTE CHANCE(CYCLIC)
```

CHANCE はデフォルトにより常に BLOCK 分散で割り付けられるが、直ちに CYCLIC 分散に再マップされることをコンパイラは知ることができなければならない。同様な所見は、TINKER や EVERS にも適用される。(EVERS は TINKER に、まばらに伸ばす方法でマップされることに注意されたい。EVERS の隣り合った要素は、TINKER の刻み幅 M で離された要素にマップされる。EVERS(1,1) は TINKER(M,1) にマップされるため、このまばらに伸ばすマッピングは、TINKER の左下隅に置かれる。)

⇑ 5.1 節には、DO ループの中で実行された場合、ループの繰返しが互いに干渉することになり、それによって、ループが INDEPENDENT であると特徴づけることを妨げる操作の一覧が示されている。その一覧には、以下の操作も加わる:

- ループの中で実行される REALIGN および REDISTRIBUTE 指示文は、同じデータへの参照や同じデータの他の再マッピングと干渉する。

  【仕様の根拠】 REALIGN および REDISTRIBUTE は、配列要素をもつプロセッサを変更することができるため、その要素の代入や引用 (use) と干渉する。同様に、同時に起る複数の再マッピング操作は、ある要素を複数の場所 (location) に配置してしまう。【以上】

8.6 再マッピングと副プログラム・インタフェース

副プログラムの仮引数が DYNAMIC 属性をもっていたならば、その副プログラムに関しては明示的引用仕様が必要である (8.14 節 参照)。REALIGN および REDISTRIBUTE 指示文の副プログラム引数インタフェースとの相互作用のルールは、以下のとおりである:

1. 仮引数は、DYNAMIC と宣言することができる。ただし、それは、結合するテンプレートを表すための、配列名の利用に関する一般的な制約を受ける。
   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、実引数の属性に依存する。仮引数と結合している実引数が、同様に DYNAMIC と宣言されていたならば、仮引数の明示的再マッピングは、手続から戻った後で呼出し元においても反映されている。RANGE 指示文 (8.11 節 参照) が、実引数で許される分散形式を限定するために使われた場合、新しいマッピングは、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。
   DYNAMIC 属性をもつ実引数と結合する仮引数は、結合する実引数が部分配列ではなく全体配列である場合に限り、*alignee* や *distributee* として REALIGN および REDISTRIBUTE に

指定することができる。
仮引数と結合する実引数が DYNAMIC と宣言されていないならば、呼出しからの戻り時に実引数の元々のマッピングに戻されなければならない。副プログラムから戻り、呼出し元が実行を再開するときに、呼出し元で参照でき DYNAMIC と宣言されていないすべての実体は、呼出し前と全く同じようにマップされている。

2. 配列またはその部分が二つまたはそれ以上のパスで参照可能な場合、それらのパスのいずれかを使って再マップすることは、HPF 規格合致ではない。たとえば、配列が実引数として引き渡された場合、副プログラムがその呼出しから戻るまでの間は、その配列を再整列することや、その配列が呼出し時に整列している配列やテンプレートを再分散することは禁止されている。これは、ぞっとするような別名問題を防止する。例を示す:

```
      MODULE FOO
      REAL A(10,10)
!HPF$ DYNAMIC ::  A
      END

      PROGRAM MAIN
      USE FOO
      CALL SUB(A(1:5,3:9))
      END

      SUBROUTINE SUB(B)
      USE FOO
      REAL B(:,:)
!HPF$ DYNAMIC ::  B
      ...
!HPF$ REDISTRIBUTE A            ! 規格合致でない
      ...
      END
```

「規格合致でない」と記された文の位置での A への代入が禁止されているのと同じ理由で、このような場合も禁止されている。一般に、別名の理由により変数への代入が規格合致でない**任意の**場面において、その変数を REALIGN や REDISTRIBUTE 指示文を使って明示的に再マップすることは、同様に禁止されている。
変数が、一つのパスでのみ参照可能であり DYNAMIC と宣言されていれば、副プログラムの中で親子結合や参照結合されていても、再マップすることが許されていることに注意されたい。そのような再マッピングは、副プログラムから呼出し元に戻った後でさえも有効である。

8.7 部分プロセッサへのマッピング

この拡張は、プロセッサが記述できる場所 (例えば DISTRIBUTE 指示文の中) に、部分プロセッサを指定することを認めることで、実体を直接的に部分プロセッサへ分散することを可能に

する。この指定された部分は、プロセッサ構成の適当な部分集合でなければならない。

拡張された *dist-target* の構文は次のとおりである。

H806 *extended-dist-target* **is** *processors-name* [(*section-subscript-list*)]
or * *processors-name* [(*section-subscript-list*)]
or *

制約: 部分配列添字並び (*section-subscript-list*) の部分配列添字 (*section-subscript*) は、ベクトル添字 (*vector-subscript*) であってはならず、添字 (*subscript*) または添字三つ組 (*subscript-triplet*) でなければならない。

制約: 部分配列添字並び (*section-subscript-list*) では、部分配列添字 (*section-subscript*) の数は、*processor-name* の次元数と等しくなければならない。

制約: DISTRIBUTE 指示文の中では、それぞれの部分配列添字 (*section-subscript*) は、宣言式 (*specification-expr*) でなければならない。

制約: DISTRIBUTE および REDISTRIBUTE 指示文の中で、*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、指定されたプロセッサ構成の添字三つ組 (*subscript-triplet*) の数と等しくなければならない。

制約: DISTRIBUTE および REDISTRIBUTE 指示文の中で、*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、指定されたプロセッサ構成の添字三つ組 (*subscript-triplet*) の数と等しくなければならない。

```
!例 1
!HPF$ PROCESSORS P(10)
      REAL A(100)
!HPF$ DISTRIBUTE A(BLOCK) ONTO P(2:5)



!例 2
!HPF$ PROCESSORS Q(10,10)
      REAL A(100,100)
!HPF$ DISTRIBUTE B(BLOCK,BLOCK) ONTO Q(5:10,5:10)
```

例 1 では、配列 A は、プロセッサ P(2) から P(5) にかけてブロック分散で分散されている。二つ目の例では、配列 B は、プロセッサ構成 Q の右下の 4 分の 1 に分散されている。

【利用者への助言】 この拡張は、独立な複数の計算段階をプロセッサの別々の部分で同時に実行することを可能にするタスキング構文 (9.4 節 参照) と一緒に使うと特に有用である。このような状況は、プログラムが並列に計算することができる複数のデータ構造を利用する場合に発生し、このとき、それぞれの独立なデータ構造の計算もさらに並列性を示す。ある種の流体力学計算で使われるマルチブロック格子における多重ブロックが一例である。ここでは、それぞれの独立なブロックは、並列性の両方のレベルを活かすために部分プロセッサ上に分散されていなければならない。【以上】

8.8 ポインタ

8.8.1 マップされたポインタ

HPF の公認拡張として、ポインタと指示先は、明示的にマップすることができる。形式的には、それぞれ 3.3 節および 3.4 節で述べた次の制約を除かなければならないことを意味する: ⇑
distributee や *alignee* は、POINTER または TARGET 属性をもつことはできない。

割付け実体の場合もそうであったように、ポインタへのマッピング指示は、すぐには効果を表さない。しかし、ポインタが割付けやポインタ代入によって指示先と結合したポインタとなったときに、役割を果たす。

明示的なマッピングをもつポインタが ALLOCATE 文で使われたとき、そのデータは指定されたマッピングで割り付けられる。

例えば:

```
      REAL, POINTER, DIMENSION(:) :: A, B
!HPF$ ALIGN B(I) WITH A(I)
!HPF$ DISTRIBUTE A(BLOCK)
      ...
      ALLOCATE(A(100))
      ALLOCATE(B(50))
      ...
      ALLOCATE(B(200))                    ! 規格合致でない
```

ポインタ A は、BLOCK 分散をもつと宣言され、一方、ポインタ B は、A と同値に整列すると宣言されている。A が割り付けられるときには、ブロック分散で生成される。B が割り付けられるときには、A の最初の 50 要素に整列する。実体が整列する先の実体がすでに生成されたり割り付けられた場合にだけ実体は整列できるため、ALLOCATE 文の順序を入れ換えてはならないことに注意されたい。また、より大きな実体 (ここでは B) は、より小さな実体 (ここでは A) に整列することはできないため、B の二番目の割付けは規格合致ではない。

明示的マッピングをもつポインタ P は、次に述べる条件の下で、ポインタ代入文により指示先 T と結合しているポインタとなることができる。

1. T のマッピングは、P のマッピングの特殊化である (特に、T は全体配列でなければならない)。さらに、
2. P が明示的に整列しているとき、その最終整列先は転写的でなく完全に指定された分散をもたなければならない。さらに、
3. P と T は、共に DYNAMIC であるか、共にそうでないかのどちらかである。

いくつかの例を示す:

```
      REAL, POINTER, DIMENSION(:,:) :: P
!HPF$ DISTRIBUTE P(BLOCK,BLOCK)
      REAL, TARGET, DIMENSION (100, 100) ::  B, C, D
!HPF$ DISTRIBUTE B(BLOCK, BLOCK)
!HPF$ DISTRIBUTE C(BLOCK, CYCLIC)
```

```
      ...
      P => B               ! 規格合致である
      P => B(1:50, 1:50)   ! 規格合致でない
                           !    : 指示先は全体配列でなければならない。
      P => C               ! 規格合致でない: 第 2 次元の分散が一致しない。
      P => D               ! 規格合致でない: D は明示的にマップされていない。
      ...
```

上記のポインタ代入 `P => B(1:50,1:50)` が規格合致でない直観的な理由は、53 ページの例 (`INHERIT A` と `DISTRIBUTE A * ONTO *`の違いを示している) が規格合致でない理由と似ている: 例えば、配列 B が、2×2 のプロセッサ構成に分散されているとしよう。このとき、部分配列 `B(1:50, 1:50)` は、すべてプロセッサ $(1, 1)$ 上に存在する。そのため、このマッピングは、P に対する `(BLOCK, BLOCK)` 分散では正しく記述できない。

次のポインタ代入文は、ポインタに対してプロセッサ構成が指定されていないにも関わらず、有効である。この例では、B のマッピングは P のマッピングの特殊化である。

```
      REAL, POINTER, DIMENSION(:) :: P
      REAL, TARGET, DIMENSION(100) ::  B
!HPF$ PROCESSORS PROC(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE P(BLOCK)
!HPF$ DISTRIBUTE (BLOCK) ONTO PROC :: B
      ...
      P => B               ! 規格合致である
      ...

      REAL, POINTER, DIMENSION(:) :: P
!HPF$ DISTRIBUTE * :: P
      REAL, TARGET, DIMENSION(100) ::  B, C
!HPF$ DISTRIBUTE B(BLOCK), C(CYCLIC)
      ...
      P => B               ! 規格合致である
      P => C               ! 規格合致である
      P => C(1:50)         ! 規格合致でない
                           !    : 指示先は全体配列でなければならない。
      ...
```

ここで、ポインタ P に対して、転写的分散を示すために、*が使われている。そのため、それぞれ BLOCK や CYCLIC で分散された指示先 B および C とポインタ結合するポインタになることができる。しかし、それでも `C(1:50)` のような部分配列を指すために使うことはできない。そうするためには、ポインタは INHERIT 属性をもたなければならない。

```
      REAL, POINTER, DIMENSION(:) :: P
!HPF$ INHERIT :: P
      REAL, TARGET, DIMENSION(100) ::  B, C
```

```
!HPF$ DISTRIBUTE B(BLOCK), C(CYCLIC)
      ...
      P => B              ! 規格合致である
      P => C              ! 規格合致である
      P => C(1:50)        ! 規格合致である
      ...
```

ポインタが転写的分散をもつことを可能にするために 3.3 節で明記された *dist-format-clause* の制約を次のように変更しなければならない (変更部分は下線を付けて示す)。

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、*distributee* が POINTER 属性をもつ場合を除き、仮引数でなければならない。

3.4 節で明記された *align-spec* への制約は、次のように変更されなければならない:

制約: ALIGN 指示文の *align-spec* が "*" で始まるときは、すべての *alignee* は、*alignee* が POINTER 属性をもつ場合を除き、仮引数でなければならない。

4.4.2 項で明記された *inheritee* への制約は、次のように変更されなければならない:

制約: *inheritee* は、*alignee* が POINTER 属性をもつ場合を除き、仮引数でなければならない。

そのような転写的マッピングをもつポインタが ALLOCATE 文で使われた場合、コンパイラは、割り付けられたデータに対して任意のマッピングを選んでよい。RANGE 宣言 (8.11 節 参照) を分散形式の集合を限定するために使ってもよい。

ポインタが DYNAMIC 属性をもっている場合、そのポインタが結合している指示先 (したがって、それは DYNAMIC 属性をもっていなければならない) は、以下の制約の範囲において、REALIGN または REDISTRIBUTE 指示文を使って再マップすることができる。

ポインタは、部分配列ではなく全体配列と結合している場合に限り、REALIGN または REDISTRIBUTE 指示文において *alignee*、 *align-target*、 または *distributee* として使ってもよい。

実体が再マップされたとき、その実体と結合しているどのポインタを介しても、新しいマッピングが反映されていることに注意されたい。

8.8.2 ポインタと副プログラム

ポインタ仮引数が明示的にマップされていない場合、実引数も明示的にマップされていてはならない。

ポインタ仮引数が明示的なマッピングをもつとき、次の一つの例外を除いて、実引数は上述したポインタ代入のルールに従わなければならない: 実引数が DYNAMIC 属性をもつ場合、対応する仮引数は DYNAMIC 属性をもつ必要はない。すなわち、153 ページの項目 3 は、次のように弱められる。

3. ポインタ仮引数が DYNAMIC 属性をもつ場合、対応する実引数は DYNAMIC 属性をもたなければならない。

RANGE 宣言 (8.11 節 参照) を、実引数の分散形式の集合を限定するために使うことができる。

ポインタ仮引数は、DYNAMIC 属性をもつことができる。この場合、実引数も DYNAMIC 属性をもたなければならない。仮引数と結合している指示先は、一つ前の節で述べた制約の範囲で、再分散されても構わない。Fortran での規則に従って、実引数が (親子結合や参照結合により) 仮引数以外を通して参照できる場合、指示先は仮引数を通してだけ再分散してもよい。仮引数が再分散されたならば、手続から戻ったとき実引数は新しいマッピングをもつ。このような場合、新しいマッピングは、実引数のレンジ制限 (もしあるならば) を守らなければならない。

8.8.3 ポインタや指示先に関する制限事項

手続 P の呼出しで、(a) 仮引数が TARGET 属性をもっている、かつ、(b) 対応する実引数が TARGET 属性をもっていてベクトル添字をもつ部分配列ではない (すなわち、実体 A または配列 A の部分である) 場合、プログラムは以下の条件を満たす場合だけが HPF 規格合致である。

1. 呼出しの間に実引数の再マッピングが起きない。または、
2. 次の制約を満たすために、プログラムの実行の残りの部分で影響がない。
   (a) 呼出しの前に A のいずれかの部分と結合しているポインタは、P の入口でポインタ結合状態が不定となり、P の実行中に再代入されない限り、出口で入口直前のポインタ結合状態にもどされる。
   (b) P の実行中に仮引数のいずれかの部分や A のいずれかの部分と結合しているポインタは、P の出口ではポインタ結合状態が不定になる。

【利用者への助言】 再マッピングが起きないことを保証する一つの方法は、仮引数に INHERIT 属性を与えることである。【以上】

【仕様の根拠】 これらの制限事項は、副プログラム境界での暗黙的な再マッピングにおいて、Fortran 規格 (F95:12.4.1.1) の次のような部分をサポートするために設けられた。

> 手続を呼び出した時、仮引数が TARGET または POINTER 属性をもっていない場合には、実引数と結合しているポインタは、対応する仮引数とは結合しない。
>
> 仮引数が TARGET 属性をもっていて、対応する実引数も TARGET 属性をもっていて、実引数がベクトル添字をもつ部分配列でない場合には:
>
> 1. 手続を呼び出した時、実引数と結合しているポインタは、対応する仮引数と結合する。
> 2. 手続の実行が終了した時点で、仮引数と結合しているポインタは、実引数との結合を保持する。
>
> 仮引数が TARGET 属性をもっていて、対応する実引数が TARGET 属性をもっていない、または実引数がベクトル添字をもつ部分配列である場合には、手続の実行が終了した時点で、仮引数と結合しているいずれのポインタも不定になる。

【以上】

本章での制限事項を表した例を示す:

```
      INTEGER, TARGET, DIMENSION (10) :: ACT
      INTEGER, POINTER, DIMENSON (:) :: POINTS_TO_ACT, POINTS_TO_DUM
!HPF$ DISTRIBUTE ACT(BLOCK)

      POINTS_TO_ACT => ACT
      CALL F(ACT)
      POINTS_TO_DUM(1) = 1             ! 不正

      CONTAINS
        SUBROUTINE F(DUM)
          INTEGER, TARGET, DIMENSION(10) :: DUM
        !HPF$ DISTRIBUTE DUM(CYCLIC)

          POINTS_TO_DUM => DUM
          POINTS_TO_ACT(1) = 1         ! 不正
        END SUBROUTINE
      END
```

POINTS_TO_DUM(1) への代入は、項目 2b に違反しているために、不正である。項目 2a に違反しているために、POINTS_TO_ACT(1) への代入も、不正である。

8.9 構造体成分のマッピング

ALIGN 指示文、DISTRIBUTE 指示文、および DYNAMIC 指示文は、構造型定義の内部で成分定義文が現れることができるところであればどこにでも現れることができる。これらの指示文に現れるすべての *alignee* と *distributee* は、その構造型定義の内部で定義される成分の名前でなければならない。構造型の成分のマッピングを可能にするために、構文規則を以下のように拡張する。

H807 *distributee-extended* **is** *object-name*
or *template-name*
or *component-name*
or *structure-component*

いずれかの成分が明示的にマップされている場合、または、いずれかの成分が明示的にマップされた型である場合、その構造型は、**明示的にマップされた型**であると言う。

制約: 構造型の成分は、その型が明示的にマップされていない場合に限って明示的に分散することができる。

制約: 構造型のデータ実体は、その構造型が明示的にマップされた型でない場合に限って明示的に分散することができる。

制約: DISTRIBUTE 指示文の *distributee* は、構造体成分 (*structure-component*) であってはならない。

制約: 構造型定義の中に現れる DISTRIBUTE 指示文の *distributee* は、構造型の成分の成分名 (*component-name*) でなければならない。

制約: 構造型定義の中に現れる DISTRIBUTE 指示文に限って、*distributee* を成分名 (*component-name*) とすることができる。

制約: REDISTRIBUTE 指示文の中でだけ *distributee* を構造体成分にすることができる。このとき、右端を除くすべての部分参照はスカラ (0 次元) でなければならない。構造体成分の右端の部分参照は、DYNAMIC 属性をもっていなければならない。

H808 *alignee-extended* **is** *object-name*
or *component-name*
or *structure-component*

制約: 構造型の成分は、その型が明示的にマップされていない場合に限って明示的に整列することができる。

制約: 構造型のデータ実体は、その構造型が明示的にマップされた型でない場合に限って明示的に整列することができる。

制約: ALIGN 指示文の *alignee* は、構造体成分 (*structure-component*) であってはならない。

制約: 構造型定義の中に現れる ALIGN 指示文の *alignee* は、構造型の成分の成分名 (*component-name*) でなければならない。

制約: 構造型定義の中に現れる ALIGN 指示文に限って、*alignee* を成分名 (*component-name*) とすることができる。

制約: REALIGN 指示文の中でだけ *alignee* を構造体成分 (*structure-component*) にすることができる。このとき、右端を除くすべての部分参照はスカラ (0 次元) でなければならない。構造体成分の右端の部分参照は、DYNAMIC 属性をもっていなければならない。

H809 *align-target-extended* **is** *object-name*
or *template-name*
or *component-name*
or *structure-component*

制約: 構造型定義の中に現れる ALIGN 指示文に限って、整列先を成分名 (*component-name*) とすることができる。

制約: *align-target* が構造体成分 (*structure-component*) であるとき、その右端を除くすべての部分参照はスカラ (0 次元) でなければならない。

上記の制約で言及されているように、構造型の定義の中で構造型の成分をマップしておくと、その型のデータ実体を生成するときに、その成分は指定したとおりにマップされる。

以下の例を考える。

```
      TYPE DT
        REAL C(100)
!HPF$   DISTRIBUTE C(BLOCK) ONTO P
      END TYPE DT

      TYPE (DT) :: S1
      TYPE (DT) :: S2(100)
```

構造型の中の配列 C は、ブロック分散で宣言されている。したがって、構造型 DT のスカラ変数 S1 がもつ構造体成分 S1%C は、プロセッサ構成 P にブロック分散される。同様に、配列 S2 のすべての要素について、成分 C はプロセッサ配列 P にブロック分散される。

構造型の定義の中の ALIGN 指示文によって、構造型の成分を同じ構造型の別の成分に対して整列させたり、他のデータ実体に対して整列させたりすることができる。構造体成分は、構造型の成分などの他のデータ実体の整列先として使用することができる。

例:

```
!HPF$ TEMPLATE T(100)
!HPF$ DISTRIBUTE T(CYLIC)

      TYPE DT
        REAL, DIMENSION(100) :: A, B, C
!HPF$   ALIGN WITH A :: B
!HPF$   DISTRIBUTE (BLOCK) :: A
!HPF$   ALIGN WITH T :: C
      END TYPE DT
```

ここでは、構造型 DT の変数は、以下のように生成される。

- 成分 B は A に対して整列する。
- A はブロック分散される。
- 成分 C はテンプレート T に対して整列する。T は構造型定義の外で定義される。

構造型の成分のマッピングの指定が不完全な場合、その成分のマッピングをどう補うかはコンパイラに任されていることに注意されたい。しかし実際には、コンパイラはそのような構造型の成分のマッピングを、変数によらずすべて同じにすると思ってよい。例えば、上のコードで成分 A の分散がなかった場合について考えてみる。B と A は整列するように宣言されているが、A の分散は与えられていない。そのような場合、構造型 DT のすべての変数は、成分 A(または言い換えると成分 B)が同じ分散となるように生成されると考えてよい。

構造型成分のマッピングに関する制約により、構造体変数のマッピングは 1 レベルだけが許されている。例として、構造型の成分が構造型であるような以下のコードを考える。

```
      TYPE SIMPLE
        REAL S(100)
!HPF$   DISTRIBUTE S(BLOCK)
      END TYPE SIMPLE

!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, *) :: HAIRY_TEMPLATE(47,73)

      TYPE COMPLICATED
        INTEGER SIZE
        REAL RV(100,100), KV(100,100), QV(47,73)
! 配列 RV, KV, QV はマップできる。
!HPF$   DISTRIBUTE (BLOCK, BLOCK):: RV, KV
!HPF$   ALIGN WITH HAIRY_TEMPLATE :: QV
        TYPE(SIMPLE) SV(100)
! SIMPLE は明示的にマップされた型なので、次の指示文は誤り。
!HPF$   DISTRIBUTE SV(BLOCK)
      END TYPE COMPLICATED

      TYPE(COMPLICATED) LOTSOF(20)

! COMPLICATED は明示的にマップされた型なので、次の指示文は誤り。
!HPF$ DISTRIBUTE LOTSOF(BLOCK)
```

ここでは、構造型の成分 SIMPLE はマップされている。このような型のデータ実体 (例えば COMPLICATED 型の SV) は、分散することができない。配列 LOTSOF も同じ理由で分散できない。

POINTER 属性をもつ構造体成分は、DYNAMIC 宣言されている場合には、REALIGN 指示文または REDISTRIBUTE 指示文で再マップすることができる。例えば以下のコードは、それ自身がブロック格子である複数のブロック (ここでは SUBGRID と呼ぶ) を割り付けてマップするために使用される。

```
!HPF$ PROCESSORS P( number_of_processors() )

      TYPE SUBGRID
        INTEGER SIZE
        INTEGER LO, HI          ! 分散先となる部分プロセッサの範囲
        REAL, POINTER BL(:)
!HPF$   DYNAMIC BL
      END TYPE SUBGRID

      TYPE (SUBGRID), ALLOCATABLE :: GRID(:)

      READ (*,*) SUBGRID_COUNT
```

```
      ALLOCATE GRID(SUBGRID_COUNT)
      DO I = 1, SUBGRID_COUNT
        READ(*,*) GRID(I)%SIZE
      END DO

! それぞれの subgrid に対する部分プロセッサを計算し、
! LO と HI の値を設定する。
      CALL FIGURE_THE_PROCS ( GRID, number_of_processors())
! それぞれの subgrid を割り付け、計算した部分プロセッサに対して分散する。
      DO I = 1, SUBGRID_COUNT
        ALLOCATE( GRID(I)%BL( GRID(I)%SIZE ) )
!HPF$   REDISTRIBUTE GRID(I)%BL(BLOCK) ONTO P( GRID(I)%LO : GRID(I)%HI )
    END DO
```

【仕様の根拠】 構造型の成分は、それが DYNAMIC 属性だけでなく POINTER 属性ももっているときに限って再マップすることができる。この制約は、HPF 指示文を使って直接に指定できないマッピングを禁止するために置かれている。例えば、以下のコードを考えてみる。

```
  !HPF$ PROCESSORS P(4)

        TYPE DT
          REAL C(100)
  !HPF$   DISTRIBUTE C(BLOCK) ONTO P
  !HPF$   DYNAMIC C                    ! 規格合致でない
        END TYPE DT

        TYPE (DT) :: S(10)
         ...
        J = 3
         ...
  !HPF$ REDISTRIBUTE S(J)%C(CYCLIC) ONTO P

          ... S(:)%C(2) ...
```

ここでは、構造型 DT の成分 C は DYNAMIC と宣言されている。したがって、配列変数 S は 10 個の配列要素から成り、それぞれの配列要素は最初はブロック分散された成分 C をもつ構造体となる。REDISTRIBUTE 指示文により、S の第 J 要素の構造体成分 C はサイクリック分散になるように再マップされる。ここで、式 S(:)%C(2) によって参照されるデータ実体のマッピングについて考えてみる。これは、配列変数 S を構成する 10 個の構造体のそれぞれから、2 番目の要素を取り出したものである。S の 1 要素 (この場合は 3 番目の要素) を再分散しているため、データ実体の他のすべての要素はプロセッサ P(1) 上に存在するが、第 3 要素だけはプロセッサ P(2) に存在する。このような分散は、HPF 指示文では直接に宣言することはできない。

Fortran 規格では、成分が POINTER 属性をもつ場合のこのような式を禁止している。すなわち、データ参照中の部分名 (*part-name*) が POINTER 属性をもつ場合、それよりも左側の部分参照 (*part-ref*) はスカラでなければならない (F95:6.1.2)。そこで、我々は以下のような規約によって上記のような状態を避ける。

- POINTER 属性をもたない構造体成分の再マッピングは禁止する。
- POINTER 属性をもつ構造体成分について、上記のような式を禁止するために、Fortran 規格の規約に従う。

【以上】

8.10 新しい分散形式

本章では、二つの新しい分散形式について述べる。構文は以下のように拡張される。

H810 *extended-dist-format* **is** BLOCK [(*int-expr*)]
or CYCLIC [(*int-expr*)]
or GEN_BLOCK (*int-array*)
or INDIRECT (*int-array*)
or *

制約: DISTRIBUTE 指示文または REDISTRIBUTE 指示文の *extended-dist-format* の中に現れる整数型配列 (*int-array*) は、整数型の一次元配列でなければならない。

制約: DISTRIBUTE 指示文の *extended-dist-format* の中に現れる整数型配列 (*int-array*) は、制限式 (*restricted-expr*) でなければならない。

制約: GEN_BLOCK 分散に現れる整数型配列 (*int-array*) の大きさは、分散先のプロセッサ構成の対応する次元の寸法と等しくなければならない。

制約: INDIRECT 分散に現れる整数型配列 (*int-array*) の大きさは、その分散が適用される *distributee* の対応する次元の寸法と等しくなければならない。

ブロック分散を「一般化」した GEN_BLOCK は、配列を不均等な大きさの連続区間に分けてプロセッサにマップさせる。各区間の大きさは利用者定義の整数型のマッピング配列の値によって指定される。つまり、マッピング配列の i 番目の要素は、分散先プロセッサ構成の i 番目のプロセッサに配置されるブロックの大きさを指定する。そのため、マッピング配列の値は負でない数に制約され、その総和は分散配列の対応する次元の寸法と比較して等しいか大きくなければならない。

マッピング配列は、DISTRIBUTE 指示文で使用される場合には制限式でなければならないが、REDISTRIBUTE 指示文では配列変数であってもよい。後者の場合、マッピング配列の値が指示文が実行された後で変化しても、分散配列のマッピングは変化しない。

ここで、l と u をそれぞれ *distributee* のある次元の下限値と上限値、MAP をマッピング配列とし、*BS(i):BE(i)* を分散先であるプロセッサ構成の対応する次元の i 番目のプロセッサ

にマップされる要素とする。すると、次式が成り立つ。

$$
\begin{aligned}
BS(1) &= l, \\
BE(i) &= \min(BS(i) + MAP(i) - 1, u), \\
BS(i) &= BE(i-1) + 1.
\end{aligned}
$$

例:

```
      PARAMETER (S = (/2,25,20,0,8,65/))
!HPF$ PROCESSORS P(6)
      REAL A(100), B(200), new(6)
!HPF$ DISTRIBUTE A( GEN_BLOCK( S) ) ONTO P
!HPF$ DYNAMIC  B
        ...
      new = ...
!HPF$ REDISTRIBUTE B( GEN_BLOCK(new) )
```

上記の宣言で、配列要素 A(1:2) は P(1) にマップされ、A(3:27) は P(2) にマップされ、A(28:47) は P(3) にマップされ、P(4) にマップされるものはなく、A(48:55) は P(5) にマップされ、A(56:100) は P(6) にマップされる。配列 B の分散は、実行時に値が計算される配列 new によって決まる。

> 【実装者への助言】 一般化されたブロック分散を使った分散配列の要素のアクセスのためには、実行時にマッピング配列の値をアクセスしなければならない。しかし、マッピング配列の大きさはプロセッサ構成の大きさと等しい (あまり大きくない) ので、ほとんどの場合にはすべてのプロセッサに複製してしまえばよい。
>
> 動的配列に対しては、マッピング配列のコピーをシステム内で個々に管理して、マッピング配列の値が変化しても分散配列のアクセスに影響を与えないようにしなければならない。【以上】

問題領域の構造が Fortran のデータ構造に直接対応付けできないような科学分野のアプリケーションは数多くある。例えば、多くの CFD アプリケーションでは、問題領域の表現に構造化できないメッシュ (2 次元での三角形、3 次元での四面体など) が使用される。このようなメッシュの格子点は一般に 1 次元配列で表現され、それらの相互の接続関係の表現にも 1 次元配列が使われる。この配列のマッピングに BLOCK や CYCLIC のような構造化された分散の機構を用いると、同じプロセッサ上に無関係な要素がマップされる結果となる。これは言い換えると、不必要な通信を大量に発生させることになる。必要なのは、関係の深い配列要素の集合を任意に同じプロセッサ上にマップさせる機構である。INDIRECT 分散は、そのような機構を提供する。

INDIRECT 分散は、データ配列のある次元の要素から分散先プロセッサ構成のある次元への多対一のマッピングを可能にする。分散する配列次元の個々の要素と分散先プロセッサの対応を指定するために、一つの整数型配列が使用される。すなわち、このマッピング配列の i

番目の要素は、i 番目の配列要素がマップされる先のプロセッサ番号を表している。マッピング配列は配列要素からプロセッサ要素へのマッピングを示すので、マッピング配列の寸法は分散される配列のその次元の寸法と一致しなければならない。また、マッピング配列の値は、分散先となるプロセッサ構成の次元の下限値と上限値の間になければならない。

マッピング配列は、DISTRIBUTE 指示文で使用される場合には制限式でなければならないが、REDISTRIBUTE 指示文では配列変数であってもよい。後者の場合、マッピング配列の値が指示文が実行された後で変化しても、分散配列のマッピングは変化しない。

例:

```
!HPF$ PROCESSORS P(4)
        REAL A(100), B(50)
        INTEGER map1(100), map2(50)
        PARAMETER (map1 = (/1,3,4,3,3,2,1,4, ..../))
!HPF$ DYNAMIC B
!HPF$ DISTRIBUTE A( INDIRECT(map1) ) ONTO P
!HPF$ DISTRIBUTE map2(BLOCK) ONTO P

        map2 = ...
!HPF$ REDISTRIBUTE B( INDIRECT(map2) ) ONTO P
        ....
```

ここで、配列 A は定数配列 map1 を使って静的に分散されている。したがって、以下のようになる。

A(1) は P(1) へマップされる
A(2) は P(3) へ
A(3) は P(4) へ
A(4) は P(3) へ
A(5) は P(3) へ
A(6) は P(2) へ
A(7) は P(1) へ
A(5) は P(4) へ
...

配列 B は動的であると宣言され、マッピング配列 map2 を使って再分散されている。

【実装者への助言】 一般に、INDIRECT 分散は、マッピング配列となる配列変数をもった REDISTRIBUTE 指示文の形で使用されることになるだろう。また、マッピング配列の大きさは分散される配列の大きさと同じなので、それ自身もほとんどの場合 BLOCK 分散などで分散されるだろう。これには、いくつかの問題点がある。この分散を正しく実装するためには、実行時システムはマッピング配列の (分散された) コピーを管理して、プログラムがマッピング配列を変更しても分散が変化しないようにしなければならない。マッピング配列として配列変数を使用することは、配列のそれぞれの要素の配置が実行時まで分からないということを意味する。したがって、指定された配列要素の配置を計算するために、通信が必要となるであろう。【以上】

8.11 RANGE 指示文

RANGE 属性は、DYNAMIC 属性または転写的な分散形式をもつデータ実体 (ポインタを含む) とテンプレートについて、あり得る分散形式を限定するために使用される。

H811 *range-directive* **is** RANGE *ranger range-attr-stuff*

H812 *ranger* **is** *object-name*
or *template-name*

H813 *range-attr-stuff* **is** *range-distribution-list*

H814 *range-distribution* **is** (*range-attr-list*)

H815 *range-attr* **is** *range-dist-format*
or ALL

H816 *range-dist-format* **is** BLOCK [( )]
or CYCLIC [( )]
or GEN_BLOCK
or INDIRECT
or *

制約: 少なくとも以下のいずれかが成り立たなければならない。

- *ranger* は DYNAMIC 属性をもつ。
- *ranger* は INHERIT 属性をもつ。
- *ranger* は DISTRIBUTE 指示文か *combined-directive* で指定され、その *dist-format-clause* は*である。

制約: *range-attr-list* の長さはそれぞれ、*ranger* の次元数と等しくなければならない。

制約: *ranger* は ALIGN 指示文または REALIGN 指示文の *alignee* であってはならない。

range-attr-list の長さはそれぞれ *ranger* の次元数と等しいので、*range-distribution* 中の *range-attr* (R とする) は *ranger* の次元 (D とする) にそれぞれ順番に対応する。つまり、次元 D は *ranger* が最終的に整列するテンプレートの座標軸 A に対応するか (この場合順番が対応するとは限らない)、あるいは、テンプレートのどの座標軸とも対応しない。

この記法を使って、*ranger* に対する RANGE 属性の意味を表現すると以下のようになる:

range-distribution-list 中の少なくとも一つの *range-distribution* について、すべての *range-attr* R は以下のどちらかである。

- 対応する座標軸 A が存在する場合、その座標軸の分散形式と適合している。
- 対応する座標軸が存在しない場合、*または ALL である。

この適合性は、*ranger* が *distributee* として現れるどんな DISTRIBUTE 指示文や REDISTRIBUTE 指示文に対しても維持されなければならない。また、*ranger* が POINTER 属性をもっていて転写的に分散されている場合、*ranger* が結合されるすべての指示先に対しても、この適合性が維持されなければならない。

ここで言う適合とは、分散形式によって以下のように規約される。

1. BLOCK は、*range-dist-format* の中の BLOCK、BLOCK() または CYCLIC() と適合する。
2. BLOCK(n) は、*range-dist-format* の BLOCK() または CYCLIC() と適合する。
3. CYCLIC は、*range-dist-format* の CYCLIC または CYCLIC() と適合する。
4. CYCLIC(n) は、*range-dist-format* の CYCLIC() と適合する。
5. GEN_BLOCK(a) は、*range-dist-format* の GEN_BLOCK と適合する。
6. INDIRECT(a) は、*range-dist-format* の INDIRECT と適合する。
7. *は、*range-dist-format* の*と適合する。

すべての分散形式は、*range-dist-format* の ALL と適合する。

combined-directive についての構文規則 H301 により、以下の形式の RANGE 指示文も許されることに注意されたい。

!HPF$ RANGE *range-attr-stuff-list* :: *ranger-list*

これは、構文規則 H801 によって定義される *combined-attribute-extended* を用いている。

例:

```
!HPF$    DISTRIBUTE T(BLOCK)
!HPF$    ALIGN A(I,J) WITH T(I)

         CALL SUB(A)
            ....

         SUBROUTINE SUB(X)
!HPF$    INHERIT X
!HPF$    RANGE X (BLOCK, *), (CYCLIC, *)
```

X の最終整列先 (この場合は継承されたテンプレート T) は 2 次元目をもたないので、X の *range-distribution* の 2 次元目には*か ALL だけを使うことができる。

```
         REAL A(100, 100, 100)
!HPF$    DISTRIBUTE A(BLOCK, *, CYCLIC)

         CALL SUB( A(:,:,1) )          ! 規格合致である
         CALL SUB( A(:,1,:) )          ! 規格合致でない
         CALL SUB( A(1,:,:) )          ! 規格合致でない
            ....
```

```
      SUBROUTINE SUB(X)
      REAL X(:, :)
!HPF$ INHERIT X
!HPF$ RANGE X (BLOCK, *)
```

サブルーチン SUB に与えられた RANGE 指示文に対して、SUB の最初の呼出しだけが規格合致である。しかし、RANGE 指示文を以下のものに置き換えるなら、この三つの呼出しをすべて規格合致にすることができる。

```
!HPF$ RANGE (BLOCK, *), (BLOCK, CYCLIC), (*, CYCLIC) :: X
```

8.12 SHADOW 指示文

隣接計算をするコード — 例えば、偏微分方程式の離散化や畳込みの計算 — においては、各プロセッサ上にローカルな部分配列を格納する領域を割り付けるときに、隣接したプロセッサから移動して来る要素のための余分なスペースを付加しておくという技巧が一般的に使われる。この余分な領域は、「シャドウ (shadow edge)」と呼ばれる。概念的に、配列の各次元に二つずつのシャドウがある。一つはローカルな部分配列の下端にあり、もう一つは上端にある。

一個の手続内であれば、コンパイラはどの配列がシャドウを必要としているかを知って、それに従って余分なスペースを確保することができる。しかし、シャドウ領域の幅は使用する計算方法に依存するので、別の手続では同じ配列に対して異なるシャドウ幅が必要とされることがある。したがって、手続をまたぐ解析がない場合、手続呼出しの度に、適切なシャドウ幅をもつスペースに配列引数をコピーしなければならないことがある。同様なデータ移動は、サブルーチンから出る時にデータを元の場所に戻すためにも必要となるだろう。この不必要なデータ移動は、必要なシャドウ幅を利用者が配列の宣言時に指定できるようにすることで避けられる。

シャドウ幅を宣言するための構文は以下のとおりである。

H817	*shadow-directive*	**is**	SHADOW *shadow-target shadow-attr-stuff*
H818	*shadow-target*	**is**	*object-name*
		or	*component-name*
H819	*shadow-attr-stuff*	**is**	(*shadow-spec-list*)
H820	*shadow-spec*	**is**	*width*
		or	*low-width* : *high-width*
H821	*width*	**is**	*int-expr*
H822	*low-width*	**is**	*int-expr*
H823	*high-width*	**is**	*int-expr*

制約: *width*、 *low-width* または *high-width* として現れる整数式 (*int-expr*) は、0 以上の値をもつ宣言式 (*specification-expr*) でなければならない。

shadow-spec に *width* を指定することは、*shadow-spec* に *width*:*width* を指定することと等価である。

```
!HPF$    DISTRIBUTE (BLOCK) :: A
!HPF$    SHADOW (w) :: A
```

このコードでは、配列 A を BLOCK で分散し、両側に幅 w のシャドウをもつことを宣言している。A が仮引数である場合、これはコンパイラに対して、手続呼出し時の不必要なデータ移動を避けるのに十分な情報を与えている。

また、一つの次元の下端と上端に異なるシャドウ幅を指定することもできる。

```
         REAL, DIMENSION (1000) :: A
!HPF$    DISTRIBUTE(BLOCK), SHADOW(1:2) ::  A
         ....
         FORALL (i = 2, 998)
            A(i) = 0.25 * (A(i) + A(i-1) + A(i+1) + A(i+2))
         END FORALL
```

このコードでは、下端には一つだけローカルでない要素が必要で、上端には二つ必要であることを宣言している。

8.13 マッピングの集合における同値と半順序

新しい分散、SHADOW 属性、および構造型の成分のマッピングが公認拡張として導入されたが、これらに対応するため 4.5 節を変更する必要がある。新しい文面は以下のとおりである (追加部分は下線を付けて示す)。

⇑

最初に、*dist-format* の指定について同値の概念を定義する。

1. 「同値である」という表現に記法 $\equiv$ を用いると、

BLOCK	$\equiv$	BLOCK	
CYCLIC	$\equiv$	CYCLIC	
$*$	$\equiv$	$*$	
BLOCK(n)	$\equiv$	BLOCK(m)	ただし、m と n は値が等しい
CYCLIC(n)	$\equiv$	CYCLIC(m)	ただし、m と n は値が等しい
CYCLIC	$\equiv$	CYCLIC(1)	
GEN_BLOCK(v)	$\equiv$	GEN_BLOCK(w)	ただし、v と w の対応する要素の値は等しい
INDIRECT(v)	$\equiv$	INDIRECT(w)	ただし、v と w の対応する要素の値は等しい

2. これら以外で、字面上異なる *dist-format* の指定は同値でない。

これは、数学的な意味で普通に用いられる同値関係である。

次に、SHADOW 属性 (文法は 8.12 節 参照) についての同値の概念を定義する。

1. *shadow-spec* の式 w_1 と w_2 は、同じ値をもつ場合、またその場合に限って同値である。
2. *shadow-spec* w は、*shadow-spec* w:w と同値である。

3. *shadow-spec* l_1:h_1 と *shadow-spec* 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、またその場合に限って同値である。
4. これら以外で、字面上で異なる *shadow-spec* 同士は同値でない。

そして、二つの SHADOW 属性は、それらの *shadow-spec-list* が要素ごとに互いに同値である場合、またその場合に限って同値であると言える。

今度は、マッピングの半順序を定義する。ここで、S (“special”) と G (“general”) をデータ実体とする。

以下のいずれかを満たす場合、またその場合に限って、S のマッピングは G のマッピングの特殊化であると言う。

1. G は INHERIT 属性をもつ。
2. S は INHERIT 属性をもたず、かつ以下のすべての制約が満たされる:
   (a) S は名前付きデータ実体または構造体成分である。
   (b) S と G は、最終整列先の形状が一致する。
   (c) S と G の対応する次元は、それぞれの最終整列先の対応する次元にマップされる。また、S と G の対応する要素は、それぞれの最終整列先の対応する要素に整列される。
   (d) 以下のいずれかである。
      i. S と G の最終整列先は明示的に分散されていない。
      ii. S と G の最終整列先は明示的に分散されている。この場合、G の最終整列先に対する分散の指示文は、以下の条件の中のいずれか一つを満たしていなければならない。
         A. *dist-onto-clause* がない。
         B. *dist-onto-clause* が “ONTO *” である。
         C. *dist-onto-clause* で指定されているプロセッサ構成の形状が、S の最終整列先に対する分散の指示文で明示されているものと一致する。

      また、以下の条件についても、いずれか一つを満たしていなければならない。
         A. *dist-format-clause* がない。
         B. *dist-format-clause* が “*” である。
         C. すべての *dist-format* が、S の最終整列先に対する分散の指示文で明示されている *dist-format-clause* の中の対応する位置の *dist-format* と、それぞれ同値 (上記で定義) である。
   (e) S と G は、どちらも SHADOW 属性をもっていないか、同値の SHADOW 属性をもっている。

8.14 明示的引用仕様を省略できる条件

4.6 節に示した条件は、DYNAMIC 属性があってもよいことを考慮して以下のように拡張される (追加部分は下線を付けて示す)。 ⇑

以下のすべての条件が満たされる場合を除いて、明示的引用仕様が必要である。

1. Fortran 規格で必要としていない。
2. 転写的に分散された仮引数はない。また、INHERIT 属性をもつ仮引数はない。
3. DYNAMIC 属性をもつ仮引数はない。
4. 対応する実引数と仮引数は、以下のどちらかである。

(a) どちらも暗黙にマップされている。
(b) どちらも明示的にマップされていて、
  i. 実引数のマッピングは、仮引数のマッピングの特殊化であり、
  ii. 実引数と仮引数の最終整列先がどちらも明示的に分散されている場合には、それぞれの *dist-onto-clause* には同じ形状のプロセッサ構成が指定されている。

5. 対応する実引数と仮引数は、
(a) どちらも順序的であるか、
(b) どちらも非順序的である。

8.15 手続の特性

SHADOW 属性と DYNAMIC 属性は、仮引数と手続の戻り値の HPF 特性になる。厳密には、4.7 節での定義は以下のように書き換えられる (追加部分は下線を付けて示す)。

- プロセッサ構成は一つの HPF 特性をもつ。それは形状である。
- テンプレートは以下の高々三つの HPF 特性をもつ。
  1. 形状。
  2. 分散。だだし、それが明示的に宣言されている場合。
  3. 分散先のプロセッサ構成の HPF 特性 (すなわち形状)。だだし、それが明示的に宣言されている場合。
- 仮データ実体は以下の HPF 特性をもつ。
  1. 整列、および整列先のすべての HPF 特性。ただし、それが明示的に宣言されている場合。
  2. 分散、および分散先のプロセッサ構成の HPF 特性 (すなわち形状)。ただし、それが明示的に宣言されている場合。
  3. SHADOW 属性。ただし、それが明示的に宣言されている場合。
  4. DYNAMIC 属性。ただし、それが明示的に宣言されている場合。
- 関数結果は仮データ実体と同じ HPF 特性をもつ。すなわち、以下の HPF 特性をもつ。
  1. 整列、および整列先のすべての HPF 特性。ただし、それが明示的に宣言されている場合。
  2. 分散、および分散先のプロセッサ構成の HPF 特性 (すなわち形状)。ただし、それが明示的に宣言されている場合。
  3. SHADOW 属性。ただし、それが明示的に宣言されている場合。
  4. DYNAMIC 属性。ただし、それが明示的に宣言されている場合。

第9章　データとタスク並列に対する公認拡張

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

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

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

9.1　活動プロセッサ集合

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

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

えてみる (これは直感的な意味を持つと言える)。

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

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

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

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

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

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

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

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

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

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

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

9.1.1 SUBSET 指示文

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

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

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

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

どちらの場合も、変更は「全体」という言葉の追加だけである。

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

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

以下は二つの形式の例である。

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

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

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

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

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

9.1.2 局所的な実体と仮引数のマッピング

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

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

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

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

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

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

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

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

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

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

9.1.3 活動プロセッサに関する他の制限

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

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

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

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

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

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

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

例を挙げる。

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

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

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

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

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

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

次の例が助けになるだろう。

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

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

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

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

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

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

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

9.2 ON 指示文

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

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

9.2.1 ON 指示文の文法

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

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

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

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

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

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

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

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

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

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

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

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

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

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

もし括弧が必要なかったら、計算はどこで行なえばよいだろうか?

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

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

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

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

9.2.2 ON 指示文の意味

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

【以上】

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

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

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

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

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

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

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

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

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

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

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

9.2.3 ON 指示文の例

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

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

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

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

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

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

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

DO i = lb, ub, $stride$

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

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

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

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

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

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

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

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

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

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

というコードは、プロセッサ p 上で、

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

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

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

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

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

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

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

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

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

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

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

上の例について、次のような変形を考える。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.2.4 副プログラムの実行に対する ON 指示文

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

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

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

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

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

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

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

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

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

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

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

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

前の例に戻る。

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

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

もし、COLLECTが次のように宣言されていたら、

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

呼出しは次のように行なわれる。

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

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

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

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

同じような例を挙げる。

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

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

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

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

次の 2 つの転写的マッピングの例も重要である。

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

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

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

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

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

⇑

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

この制約は、次のように読み替えることができる。

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

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

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

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

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

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

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

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

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

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

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

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

9.3 RESIDENT 節、指示文、構文

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

次の例を考えてみよ。

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

この指示文によって、次の事実が表明されている。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

【以上】

9.3.1 RESIDENT 節の例

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

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

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

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

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

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

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

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

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

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

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

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

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

K のループにおいて、次の参照はローカルである。

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

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

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

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

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

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

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

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

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

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

9.3.2 RESIDENT 指示文の手続引用への適用

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.4 TASK_REGION構文

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

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

9.4.1 TASK_REGION構造の文法

タスク範囲とは、一対の注釈によって区切られる単一の入口を持つ範囲である。

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

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

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

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

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

9.4.2 TASK_REGION構文の意味

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

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

タスク範囲の内側では、以下の制限が守られなければならない。

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

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

9.4.3 実行モデルと使用法

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

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

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

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

9.4.4 実装

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

9.4.4.1 ローカル化された計算と通信

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

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

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

9.4.4.2 複製された計算

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

9.4.4.3 入出力の持つ意味

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

9.4.4.4 SPMD コード 生成か MIMD コード 生成か

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

9.4.5 例: 2-D FFT

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

基本となる逐次版の 2D FFT コードを以下に示す:

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

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

```
        END DO
```

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

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

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

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

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

            IF (done1) EXIT
            a2 = a1

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

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

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

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

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

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

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

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

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

```
            CALL colffts(a2)

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

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

第10章　非同期入出力に関する公認拡張

本章では、HPF プログラムまたは Fortran プログラムから、非同期入出力を実行するための機構を定義する。これは、Fortran95 規格草案 X3J3/96-007r1 への変更として示される。この拡張は、X3J3 非同期入出力拡張提案である論文 X3J3/96-158r2 の一部である。一言でいえば、この拡張は、書式無し直接データ転送を、プログラムの実行とは非同期的に実行できるようにするためのものである。データ転送の完了を待ち合わせるためには、WAIT 文を使用することができる。データ転送が完了したかどうかを判定するためには、INQUIRE 文を使用することができる。

第 9.3.4 項の規則 R905 *connect-spec* に、以下を追加する。

or　ASYNCHRONOUS

第 9.3.4.10 項の後に、「**OPEN 文の ASYNCHRONOUS 指定子**」というタイトルの新しい項を追加する。これは、以下のような段落を含む。

> OPEN 文中の装置に対して ASYNCHRONOUS 指定子が指定されている場合、その装置に対する READ 文及び WRITE 文の制御情報並び中に、ASYNCHRONOUS 指定子を指定してもよい。
>
> READ 文又は WRITE 文中に ASYNCHROUNOUS 指定子が指定された場合、プロセッサは、データ転送を非同期的に実行してもよい (ただし、必ずしもその必要はない)。WAIT 文は、非同期入出力処理を待ち合わせるため、又は非同期入出力処理の状態を問い合わせるために用いられる。

第 9.4.1 項の規則 R912 *io-control-spec* に以下を追加する。

or　ID = *scalar-default-int-variable*
or　ASYNCHRONOUS

さらに、以下の制約を追加する。

制約: ASYNCHRONOUS 指定子又は ID=指定子のどちらか一方を指定した場合、その両方を指定しなければならない。

制約: ASYNCHROUNOUS 指定子を指定した場合、REC=指定子を指定しなければならず、かつ「書式」を指定してはならず、かつ「変数群名」を指定してはならない。

制約: ASYNCHRONOUS 指定子を指定した場合、データ転送文中のどの式においても、関数を引用してはならない。

第 9.4.1 項の終わりに、以下の段落を追加する。

ID=指定子を追加することにより、非同期データ転送が開始される。データ転送文が ID 変数に返す値と同じ値の ID が指定された WAIT 文が、データ転送文の実行後、最終的に実行されなければならない。この WAIT 文を、「対応する」WAIT 文と呼ぶ。非同期データ転送は直接かつ書式無しでなければならないことに注意されたい。

対応する WAIT 文は、非同期データ転送文が実行されたのと同じ副プログラムの同一の分身 (instance) で実行されなければならない。

【実装者への助言】 この制約は、コンパイラによる、WAIT 文をまたぐコード移動最適化を防ぐためのものである。非同期入出力並び中に指定された変数に関連するいかなる処理も、対応する WAIT 文が実行されるまでに行ってはならない。【以上】

io-unit を開く時に、ASYNCHRONOUS 指定子が省略された場合、ASYNCHRONOUS 指定子も ID=指定子も指定してはならない。

第 9.4.1 項の制約の後の、4 番目と 5 番目の段落中に現れる「IOSTAT=指定子、又は SIZE=指定子」を、「IOSTAT=指定子、SIZE=指定子、又は ID=指定子」に変更する。

以下の文章を、第 9.4.3 項の最後の段落の前に挿入する。

非同期データ転送の場合、誤り条件はデータ転送文の実行中か、後続するデータ転送の間に発生する。誤り条件がデータ転送文中で発生し、その結果、プログラムが終了しなかった場合、誤り条件はデータ転送文中の ERR=指定子と IOSTAT=指定子によって検出できる。誤り条件が後続するデータ転送の間に発生し、その結果、プログラムが終了しなかった場合、誤り条件は、対応する WAIT 文中の ERR=指定子と IOSTAT=指定子によって検出できる。

第 9.4.3 項の最後の段落の、「実行」の最初の出現を、「実行と後続するデータ転送」に変更する。

第 9.4.4 項に、以下の文章を追加する。

非同期データ転送では、手順 1–8 は、非同期データ転送文と対応する WAIT 文の両方に対応する。手順 4–7 は、プログラムの実行とは非同期的に実行される可能性がある。非同期データ転送をサポートしない実装では、手順 1–8 は、非同期データ転送文によって実行される。この場合でも、対応する WAIT 文は、状態情報を返すことがただ一つの効果であるように実行されなければならない。

非同期データ転送文中に、「入力項目」又は「出力項目」として指定された任意の変数、又はそのような変数と結合している任意の変数は、対応する WAIT 文が実行されるまで、参照したり、定義状態又は不定状態にしてはならない。ただし、ポインタは、そのような変数と結合してもよい。

未完了の非同期データ転送処理 (READ 文又は WRITE 文) が複数あってもよい。ただし、2 つの WRITE 処理に対して、間に WAIT 文をはさむことなく同一の装置番号や記録番号を指定してはならない。

【利用者への助言】 HPF では、非同期であるかどうかに関わらず、READ 文の入出力並び中で、左の項目が右の項目を定義してもよい。これは、次のような文、

```
READ(10,ID=IDNUM,REC=10) I,A(I)
```

が規格合致であり、同期 READ 文と同じ入力動作となることを意味する。【以上】

第 9 章では、「及び INQUIRE 文」を、「INQUIRE 文、及び WAIT 文」に変更する。
第 9.6.1.14 項の段落の最後の文として、以下の文を追加する。

指定された装置に対して未完了のデータ転送処理がある場合、NEXTREC=指定子に代入される値は、全ての未完了のデータ転送が既に完了しているかのように、その発行順に計算される。

第 9.6.1 項の規則 R924 *inquire-spec* に、以下を追加する。

or ID = *scalar-default-int-variable*
or PENDING = *scalar-default-logical-variable*

さらに、以下の制約を追加する。

制約: INQUIRE 文中に FILE=指定子を指定した場合、ID=指定子や PENDING=指定子を指定してはならない。

制約: ID=指定子又は PENDING=指定子のどちらか一方を指定する場合、その両方を指定しなければならない。

第 9.6.1.22 項の後に、「**INQUIRE 文の ID=指定子と PENDING=指定子**」というタイトルの新しい項を追加する。これは、以下のような段落を含む。

INQUIRE 文中に ID=指定子が指定された場合、PENDING=指定子中に指定された変数の値が真になるのは、ID=指定子で識別される、指定された装置に対するデータ転送が、未完了の場合だけである。それ以外の場合には、PENDING=指定子中に指定された変数の値は偽になる。

10.1 WAIT 文

H1001 *wait-stmt* **is** WAIT (*wait-spec-list*)

H1002 *wait-spec* **is** UNIT = *io-unit*
or ID = *scalar-default-int-expr*
or ERR = *label*
or IOSTAT = *label*

制約: *wait-spec-list* には、UNIT=指定子をちょうど 1 つ、ID=指定子をちょうど 1 つと、他の各指定子を高々1 つ、指定しなければならない。

WAIT 文により、非同期データ転送が完了する。IOSTAT=指定子と ERR=指定子は省略可能であり、それらは、それぞれ第 9.4.1.4 項と第 9.4.1.5 項で説明されている。

【実装者への助言】 実装者は、任意の又は全ての非同期入出力を同期的に実行してもよい。このような場合は本質的に、データ転送文の転送結果を記録するために ID=節を使用し、その結果を対応する WAIT 文に渡すことを意味している。【以上】

第11章　HPF外来機能に関する公認拡張

この章の内容は、第6章に基づくものであり、LOCALやSERIAL (第11.1節) といった様々な並列処理モデル、及びHPF (第11.3節)、 C (第11.4節)、 Fortran (第11.5節)、Fortran 77 (第11.6節) といった様々な言語の、HPFに対する特別な引用仕様が定義されている。外来モデルの手続から利用可能なライブラリ手続は、第11.7節で定義されている。これらの引用仕様の定義は、Fortran言語による。C言語から引用できる同様のライブラリを定義して、実装してもよい。HPFの外来機能は、相互引用の機能を一般化するために定義されたものである。これらを、ここでは扱われていない引用仕様に対するモデルとして利用してもよい。いくつかの外来引用仕様が、付録E以降でさらに追加される。

HPF1.1では、 HPF、HPF_LOCAL、 及び HPF_SERIAL という特別な外来種別が定義されていた。また、F90_LOCAL に関する議論も少し述べられていた。本規格による、より一般的な仕様においても、互換性を保つために、HPF というキーワードは引続き採用されている。第 6 章で定義されているように、HPF のモデルはグローバルである。HPF_LOCAL は、LANGUAGE='HPF',MODEL='LOCAL' と同じであり、HPF_SERIAL は、LANGUAGE='HPF',MODEL='SERIAL' と同じである。F90_LOCAL 外来種別は、本規格では、LANGUAGE='FORTRAN',MODEL='SERIAL' 外来種別となった。

手続の呼出し側から見ると、「グローバル」 HPFプログラムからの 外来手続の引用は、通常の手続引用と同じ意味をもっている。呼ばれ側手続は、通常の手続とは異なった意味をもち得る。以下の節では、様々な外来種別のオプションをもつ呼ばれ側手続を、コーディングする際の規約を述べる。ある特定の HPF コンパイラでサポートされる 外来種別のオプションは、処理系依存である。それらは、この章で述べられているものに限定されない。さらに、外来副プログラムは、必ずしもHPFコンパイラで翻訳する必要はない。また、実行する外来プログラム単位のコードが、その言語用に定義された規約に従っているなら、LANGUAGE句に指定された言語のコンパイラである必要もない。これらの引用仕様は、移植性を高め、相互引用を促進するために定義されたものだが、実装者やプログラマは、その他のモデルや言語の組合せを定義してもよい。

11.1　様々な外来モデル : ローカルモデルとシリアルモデル

グローバルHPFプログラムでは、プロセッサの集合は、単一論理スレッドのプログラム制御上で、時には同期を取りながら、協調して動作している、と考えられる。第6章では、グローバルHPFから引用できる2つのモデルが定義された。その一つは、LOCALモデルである。このモデルは、単一プロセッサ「ノード」用コードであり、全ての活動プロセッサが実行に参加するが、それらは、物理プロセッサにマップされているデータだけを直接参照することができる。もう一つは、 SERIALモデルである。このモデルでは、単一のプロセッサだけが実行に参加する。必要な全てのデータは、シリアル副プログラムが起動される前に、呼出し側

によって、実行プロセッサに集められる。それぞれの利用例を挙げると、 LOCAL モデルは、データ転送のためのメッセージ通信が明示されているような低水準コードによるプログラムの場合に有効である。一方、SERIAL モデルは、システムライブラリや特別な I/O 手続を引用するために必要となる可能性がある。

LOCAL モデルも SERIAL モデルも、グローバル HPF プログラムから引用することができるが、一般的には、これらのモデルの引用を混合してはならない。ローカル手続やシリアル手続から グローバル HPF 手続を引用してはならない。また、ローカル手続から シリアル手続を引用してはならない。第 6.3.1 項では、様々なモデル同士の関係がどの様なものであるかを詳細に述べている。

グローバル HPF から引用される全てのローカル副プログラムとシリアル副プログラムには、以下のような制約が追加される。

- グローバル HPF から直接引用される副プログラムは、再帰的ではあってはならない。
- グローバル HPF から直接引用される副プログラムは、選択戻りを使用してはならない。

ローカル副プログラムやシリアル副プログラム内の I/O 文の動作は処理系依存である。

11.1.1 ローカルモデル

外来手続を、ローカル手続であると指定すれば、その手続が、各プロセッサ上で実行するためのコード (SPMD コード) であることを明示できる。この項では、呼出し側と EXTRINSIC(MODEL='LOCAL') 呼ばれ側手続との間の規約を述べる。外来手続 (HPF プログラムから引用される概念的な単一の手続要素) と個々のローカル手続 (各ノード上で実行される) とを、混同しないようにすることは重要である。外来手続の「起動」により、各プロセッサ上で独立に、ローカル手続が起動される。外来手続の「実行」は、各実行プロセッサ上でのローカル手続の並列実行から構成される。各ローカル手続は、RETURN 文の実行によっていつでも終了することができる。しかし、外来手続は、全てのローカル手続が終了して初めて、全体として終了する。つまり、プロセッサは、グローバル HPF 呼出し側に戻る前に、同期をとる。

ローカル手続から グローバルな呼出し側に戻る時を除いては、ローカルに実行しているプロセッサが暗黙的に同期をとる必要は全くない。ローカル手続内で宣言された変数は、各プロセッサの非公開なローカル記憶領域に割り付けられる。プロセッサ外のデータを参照するためには、ローカルモデルコードの実行前に、データをプロセッサにコピーするための通信を発行するか、若しくは独立に実行中のローカル手続のコピー間で、明示的に通信処理を行う必要がある。個々の実装では、例えば、メッセージ通信ライブラリや、共有メモリ機構など、処理系依存の通信手段を提供してもよい。そのような通信手段は、この規格の範囲外である。しかし、通信の機能がなくても、制御構造の独立性のみを要求するような多くの有益な移植性の高いアルゴリズムで、ローカル手続を利用できる、ということに注目されたい。

LOCAL モデルで仮定されているのは以下のことだけである。すなわち、非順序的な配列の次元が、直方体プロセッサ格子の次元に独立に対応づけられ、さらに配列の各次元は、プロセッサの高々1 つの次元に対応づけられ (「歪んだ」マッピングはない)、かつ配列の 2 つの次元がプロセッサの同じ次元に対応づけられることはない、という制約である。

この制約は、各物理プロセッサが所有する配列要素部分集合のローカルな構成を、直方体にできるための十分条件である。(もちろん、ある要素のローカルな添字が与えられた時に、そのグローバルな添字を計算することや、その逆変換は、複雑な計算であり得る。しかし、それは可能である。) サイクリック分散の場合、配列の一部分が複数個、ローカルなプロセッサにマップされ得る。

この節で述べられた規約に従わない外来種別を実装する場合、そのモデル名やキーワードは、LOCAL という語を含まないよう推奨する。

11.1.1.1 ローカル副プログラムの呼出し規約

スカラ仮引数、スカラ関数結果、及び順序的な配列は、各物理プロセッサ上に複製される。このようなマッピングを、引用仕様内で明示しても良い。(順序的な配列の場合は除く。これらは、明示的にマップしてはならない。) しかし、複製以外のマッピングを指定することは、HPF 規格合致でない。仮引数は、明示的にマップされた構造型であったり、明示的にマップされた構造体成分をもっていてはならない。

非外来副プログラムを引用する場合と同様に、実引数はどの様にマップされていてもよい。もし必要なら、外来手続引用の前後で自動的に、HPF 処理系が、実引数を正しくマップされた一時領域にコピーする。

ローカル副プログラムを引用するための規約は、 GLOBAL モデルと LOCAL モデル間の引用仕様にだけ適用されることに注意されたい。この規約は、LOCAL モデル手続からさらに呼出される副プログラムに対しては適用されない。

11.1.1.2 ローカル手続の呼出し手順

外来ローカルプログラムの実行は、以下に述べられた処理が、各プロセッサ上でのローカル手続の起動に先立って行われる「かの如くに」実装されなければならない。(関連する規約のリストと「かの如くに」の意味は、 第 6.3.2 項を参照のこと。) ここで要求されている任意の処理は、プログラマの責任ではなく、呼出し側プログラム単位のコンパイラによって行われるものであり、それらはローカル手続には影響を与えない。

1. プロセッサは同期をとる。言い換えれば、論理的に呼出しより前の全ての動作が完了する。
2. 各実引数は、外来手続に対して宣言された引用仕様内の (明示的又は暗黙的な) 指示文に従って、必要なら再マップされる。この場合、引用仕様内の HPF マッピング指示文は、強制力がある。つまり、コンパイラは、ローカル外来手続を起動する際に、これらの指示文に従わなければならない。(この規則が設けられた理由は、データのマッピングが、ローカル手続内であからさまに参照できるからである。) 順序的な配列やスカラ仮引数に対応する実引数は、全てのプロセッサ上に、(例えば、ブロードキャストによって) 複製される。明示的にマップされた構造体成分をもつ構造型や明示的にマップされた構造型のスカラを、グローバル HPF から引用されるローカル外来手続の引数に指定してはならない。
3. 呼ばれ側手続から参照可能な変数が複製されている場合、その全てのコピーは、原始プログラムを逐次的に実行した場合の、現在の正しい値をもつように、呼出し前に更新される。

これらが実行された後、ローカル手続が各プロセッサ上で起動される。ローカル手続起動中に利用できる情報は、以下の第 11.1.1.3 項で述べられる。

外来ローカル手続の実行が完了して、呼出し側に制御が戻った時、以下の処理が、既に行われた「かの如くに」呼出し側の実行が再開されなければならない。

1. 全てのプロセッサは戻り時に同期をとる。言い換えれば、呼出し側の実行再開前に、ローカル手続の全てのコピーの実行が完了している「かの如くに」、グローバルプログラム単位内の演算は実行される。
2. 必要ならば、引数 (と外来関数の結果) のマッピングが、呼出し前のマッピングに戻される。

【実装者への助言】 ローカル副プログラムから戻る前に、複製された変数が副プログラム内で同じ値に更新されたことを確認するように実装してもよい。しかし、そのように実装しなければならない訳ではないし、奨励すらされないだろう。これは、ローカル副プログラム側の責任であり、呼出し側の確認はどのようなものでも、スピードと、例えばデバッグのしやすさとのトレードオフとなる。【以上】

11.1.1.3 ローカル手続内で利用可能な情報

呼出し側が (グローバル) 外来手続引用仕様に対して渡す、各「グローバル引数」に対応して、各プロセッサ上で起動されるローカル手続には、「ローカル引数」が渡される。各グローバル引数は、HPF の配列又はスカラである。それに対応するローカル引数は、グローバル配列内のローカルに割り当てられている部分か、スカラ引数のローカルコピーか、全プロセッサに複製された順序的な配列である。HPF の配列が複製されている場合、各ローカル手続は、実引数全体のコピーを受け取る、ということに注意されたい。HPF 呼出し側が渡す実配列引数を、「グローバル配列」と呼ぶ。グローバル配列の部分格子であり、(そのプロセッサに割り当てられている、という理由で) ローカル手続の 1 つのコピーに渡される引数を、「ローカル配列」と呼ぶ。

外来手続が関数の場合、ローカル手続もまた関数であり、その返値はスカラでなければならない。各ローカル関数の返値は、同一でなければならない。

グローバル HPF プログラムが、実配列引数 `X` をもつローカル副プログラム `A` を呼出し、`A` が、配列 `X` の一部を仮引数 `P` として受け取った場合、`A` は、`P` 又は `P` の一部を実引数として、別のローカル副プログラム `B` を呼び出すことができる。

各ローカル関数内で、各ローカル引数に関して、対応するグローバル引数のマッピングを知り、グローバルな添字からローカルな添字への変換及びその逆変換ができるように、実行時のインタフェースにより、十分な情報が提供されなければならない。このような情報を提供する特別な一連の手続が、第 11.7.1 項に、HPF ローカルライブラリとして述べられている。

このような情報をローカルプログラム単位から利用できるようにする手段は、実装と、ローカルプログラム単位に利用されるプログラミング言語に依存する。

11.1.2 シリアルモデル

この節では、副プログラムの分身が (一つ以上かもしれないプロセッサのうち) 一つのプロセッサだけで実行されるコードを書くための規約を定義する。プログラム単位の外来モデルが `SERIAL` の場合、HPF コンパイラは、その副プログラムが、単一プロセッサ上で実行され

るようにコーディングされている、と仮定しなければならない。グローバル HPF 呼出し側からみると、SERIAL 手続は、同様にコーディングされた HPF 手続と同じように振舞う。それらの違いは、(性能のような) 処理系依存の動作だけである。

今のところ、どのプロセッサが HPF_SERIAL 手続を実行するのかを指定する方法はない。

11.1.2.1 シリアル手続呼出し手順

グローバル HPF から SERIAL 手続を起動する前に、以下の処理が行われた「かの如くに」、プログラムは動作する。

1. プロセッサは同期をとる。論理的に呼出しより前の全ての動作が完了する。
2. 全ての実引数は、実際に SERIAL 手続を実行するプロセッサに再マップされる。SERIAL 手続は、各引数を順序的な引数とみなす。

SERIAL 手続は、一つのプロセッサ上で実行されている「かの如くに」動作する。グローバル HPF から起動された SERIAL 手続の分身の実行完了後、以下の処理が行われた「かの如くに」、プログラムは動作する。

1. 全てのプロセッサが、呼出しの後に同期をとる。
2. 実引数のマッピングが、呼出し前のマッピングに戻される。

11.2 外来言語との結合仕様

前節では、HPF の外来機能に対して定義されている実行モデルに関して、規約と考察を述べた。HPF の外来引用仕様は、引用された副プログラムに使用されている言語の結合仕様を、コンパイラに伝える目的でも利用される。ここでは、4 つの言語との結合仕様が定義されている。それらは、HPF、 Fortran、 F77、及び C である。実装者は、他の引用仕様をサポートしてもよいし、ユーザーが引用仕様をカスタマイズできるようにしてもよい。この節では、特別な外来種別である HPF_LOCAL と HPF_SERIAL を定義する。

言語引用仕様の重要な機能は、明示的な外来引用仕様における、仮引数の属性の拡張である。これにより、プログラマは、外来種別の異なる手続間で、引数を渡す際の全ての要素を制御できる。この機構は、 C と Fortran77 に対する引用仕様で多用されるが、他の言語の引用仕様にも適用可能なので、ここでより一般的な文脈において定義されている。

11.2.1 引数の制御

外来種別の手続の仮引数に対する特別なデータ属性として、MAP_TO、LAYOUT、及び PASS_BY がある。これらは、適切な外来種別の手続に対する明示的引用仕様内で、仮引数の型宣言文中にのみ指定できる。特に、これら全ての属性は、LANGUAGE ='C' (第 11.4 節) という種別の外来引用仕様に対して定義されており、後の 2 つの属性は、 LANGUAGE ='F77' (第 11.6 節) という種別の外来引用仕様に対して定義されている。

この言語拡張の目的は、異なるプログラミング言語で記述された手続間で、引数の受け渡しを可能にするための EXTRINSIC 引用仕様を、より柔軟な機構にすることである。これら 3 つの属性は、様々な引数渡しの規約やオプションを考慮したものであり、またほとんど等価なデータ型表現や配列の配置方法の間で、引数渡しを行う便利な方法を提供している。しか

し、これらの機構は、様々な言語のデータ型や実装間で完全な等価性を提供する、という問題を解決するものではない、ということに注意されたい。

MAP_TO 属性は、本質的には同じだが、必ずしも同じ値の範囲や等価なマシン表現をもたない (異なる言語の) データ型間における、引数渡しの標準的、汎用的な機構を提供するために設計された。実引数の値が、新たなデータ型で適切に表現されるかどうか、また、最初に新たな言語へ変換され、外来手続内で演算が行われ、そして元の言語へ暗黙的に逆変換される、という一連の操作によって、その値が変化するかどうか、ということに関しては、(言語の実装者ではなく) プログラマに責任がある。LAYOUT 属性は、異なる言語の手続間で、引数として配列を渡す時に、一つ又はそれ以上のプロセッサ内での配列要素順序を変える必要がある場合に使用される。最後の PASS_BY 属性は、言語の実装毎の差異を考慮して、引数渡しの機構をより詳細に制御できるように設計されており、それにより、HPF 以外の言語で提供されている引数渡しのオプションを選択したり、HPF のマッピング情報を、データの値と一緒に HPF 以外の手続へ渡したい時に、処理系依存のデータ構造を渡すことができる。

これらの属性は、 Fortran 規格の *attr-spec* に関する構文規則 R503 を拡張することにより定義されている。*type-declaration-stmt* の規則 R501 は、拡張された *attr-spec* を参照することを除き変更されない。以下の最初の 2 つの制約は、Fortran 規格からの変更はないが、確認のため改めて述べる。というのは、それらは全ての属性に適用される一般的なものであるからだ。Fortran 規格の規則 R501 から R506 までの残りの制約は、規格中で既に定義されている属性に特有のものであり、ここでもそれは仮定されているが、繰り返さない。このような Fortran 構文に対する変更は、言語間の相互引用性が Fortran 社会全体の興味の対象であるため、次改訂で言語規格に採用されることを期待して行われた。

H1101 *type-declaration-stmt-extended* **is** *type-spec* [[, *attr-spec-extended*] ... ::] *entity-decl-list*

H1102 *attr-spec-extended*
is PARAMETER
or *access-spec*
or ALLOCATABLE
or DIMENSION (*array-spec*)
or EXTERNAL
or INTENT (*intent-spec*)
or INTRINSIC
or OPTIONAL
or POINTER
or SAVE
or TARGET
or MAP_TO (*map-to-spec*)
or LAYOUT (*layout-spec*)
or PASS_BY (*pass-by-spec*)

H1103 *map-to-spec* **is** *scalar-char-initialization-expr*

H1104 *layout-spec* **is** *scalar-char-initialization-expr*

H1105 *pass-by-spec* **is** *scalar-char-initialization-expr*

制約: 同じ *attr-spec-extended* を、一つの *type-declaration-stmt* 中で 2 回以上指定してはならない。

制約: 一つのデータ要素には、一つの有効域内で、どの属性も 2 回以上明示的に指定してはならない。

制約: MAP_TO 属性、LAYOUT 属性、及び PASS_BY 属性は、これらの属性が明示的に定義されている外来種別の有効域内で、仮引数に対してだけ指定できる。

F95:12.2.1.1 で定義され、第 8.15 節で拡張された**仮データ実体の特性**の定義は、仮データ実体の MAP_TO 属性、 LAYOUT 属性、及び PASS_BY 属性を含むように、さらに拡張される。

MAP_TO 属性における *map-to-spec* の値には、名前付き実引数のデータ型を、外来手続内の仮引数のデータ型にどのように変換するかを指定する。その例は、第 11.4.2.1 項で示される。

MAP_TO 属性を指定できる外来種別に対しては、*map-to-spec* として許される値の集合が定義される。型と変換先の型との間で、許される値の範囲が異なり、実引数又は実引数の部分実体の値が、変換先の型で許される範囲にない場合、それと結合する仮引数又はその部分実体の値は不定となる。逆に、仮引数や仮引数の部分実体の値が、それと結合する実引数の値として許される範囲内になく、実引数が変数の場合、結合する実引数又は実引数の部分実体の値は、不定となる。

仮引数の型と仮引数の変換可能な型との間で、精度、表現方式、許される値の範囲、又は記憶列の不一致がある場合、外来手続の引用中、仮引数の表現が、変換可能な型の実体に対する呼ばれ側手続の想定に適合することを、コンパイラは保証する。手続からの戻り時には、コンパイラは、変数である実引数の値が、宣言された型と種別に戻ることを保証する。

【利用者への助言】 この規則は、相互引用機能の移植性を保証するために定められた。しかし、大きな実体の場合、データ型に用いられる表現方式と変換可能な型に用いられる表現方式が一致していない場合、大きなオーバヘッドが生じる可能性があることに注意すべきである。【以上】

LAYOUT 属性において、ある外来引用仕様に対して許される *layout-spec* の値により、名前付き実引数のデータ配置から、外来手続の仮引数のデータ配置への変換方法を指定できる。その一例が、第 11.6.3 項で示される。仮配列引数に対する LAYOUT 属性が省略されている場合、データ配置は、他の暗黙のデータ配置が、与えられた外来種別に対して定義されていない限り、同じモデルの HPF 手続へ渡された場合と同じになる。

PASS_BY 属性において、ある外来引用仕様に対して許される *pass-by-spec* の値により、名前付き実引数を、外来手続内の仮引数と結合する際に使用される機構を選択できる。その例が、第 11.6.3 項 と第 11.4.2.1 項で示される。PASS_BY 属性が省略された場合、引数結合の機構は、外来言語処理系に関するコンパイラの知識に基づくものとなり、それは処理系依存である。

11.3 HPF 言語との結合仕様

プログラム単位は、既定値では、HPF 言語で記述されているとみなされる。HPF 言語の場合、明示的引用仕様内で、MAP_TO 属性、 LAYOUT 属性、又は PASS_BY 属性を指定する必要はない。副プログラムを結合するのに必要な全ての情報は、Fortran 規格の明示的引用仕様から得ることができる。

Fortran 規格の第 14.7 節で述べられている規則は、Fortran 言語に基づく SERIAL 及び LOCAL プログラム単位の有効域で宣言された変数に適用される。特に、ある変数の定義状態、結合状態、割り付け状態が、Fortran 副プログラムの RETURN 文や END 文の実行時に定義されている場合、SERIAL 副プログラムや LOCAL 副プログラム内のそのような変数もまた、RETURN 文や END 文の実行時に定義される。

異なるモデルの外来副プログラムで行われる I/O や、グローバル HPF 内で使用されるファイル名及び装置番号と、ローカル又はシリアル副プログラムコード内で使用されるそれらとの対応は、処理系依存である。

11.3.1 HPF_LOCAL に関する特記事項

ローカルなプロセッサ毎の手続を作成する際、どのような HPF の機能を利用できるかに関して考察する。

ローカルプログラム単位は、REDISTRIBUTE 指示文と REALIGN 指示文を除く、全ての HPF 構文を使用できる。ただし、DISTRIBUTE 指示文、 ALIGN 指示文、及び INHERIT 指示文は、仮引数にだけ適用できる。すなわち、全ての *alignee* と *distributee* は、仮引数でなければならず、全ての *align-target* は、テンプレートか仮引数でなければならない。ローカル HPF プログラム単位内のマッピング指示文は、それらが グローバル HPF コード内で指定されたかのように、グローバル配列に対して適用されていて、その一部が各プロセッサに渡されている、というグローバルな意味をもっているものとみなされる。(そのようなマッピング指示文が主として使用されるのは、グローバル HPF モジュールから引用される HPF_LOCAL モジュール内である。)

HPF_ALIGNMENT、 HPF_TEMPLATE、及び HPF_DISTRIBUTION マッピング問合せライブラリサブルーチンは、非順序的なローカル配列に適用することができる。それらの結果は、単一ノードに全ての要素が割り当てられているグローバル配列の場合と同じである。

第 6.3.1 項で述べたように、ローカル HPF プログラム単位は、実引数を通して参照可能なデータ以外のグローバル HPF 内のデータを、直接又は間接に参照してはならない。特に、ローカル HPF プログラム単位は、 グローバル HPF 内の COMMON ブロックを参照できない。すなわち、ローカル HPF プログラム単位に現れる COMMON ブロックは、グローバル HPF 手続の COMMON ブロックとは異なる。同じ実行プログラムにおいて、ローカル HPF プログラム単位の COMMON ブロックと HPF プログラム単位の COMMON ブロックに、同じ名前を指定してはならない。

ローカル副プログラム内の COMMON ブロックは、ローカル副プログラム内の局所変数と同様に、各プロセッサにローカルな記憶領域に割り付けられたローカルデータである。この記憶領域は、ローカルにしか参照できず、一般に、各プロセッサ上で異なる値をもつ。さらに、ローカルな COMMON ブロックの大きさは、各プロセッサ毎に異なる可能性もある。

Fortran 規格によれば、有効域外の COMMON ブロックは、SAVE 属性をもたない限り保存されない。これは、ローカルな COMMON ブロックの場合も同様であり、ローカル副プログラムの呼出し間で値を保存したい場合には、SAVE 属性を指定しなければならない。

明示的にマップされた構造型スカラを、HPF_LOCAL 副プログラムの引数に指定してはならない。

仮引数の属性 (型、種別、次元数、省略可能性、授受特性) は、明示的引用仕様内の対応する仮引数の属性と一致していなければならない。EXTRINSIC('HPF','LOCAL') 手続の仮引数に、手続名を指定してはならない。

EXTRINSIC('HPF','LOCAL') 手続の仮引数に、POINTER 属性を指定してはならない。

EXTRINSIC('HPF','LOCAL') 手続の非順序的な仮配列引数は、引用仕様内で形状が明示してある場合でも、形状引継ぎ配列でなければならない。仮配列引数の形状は、実行プロセッサが 1 台でない限り、一般に、対応する実引数の形状とは異なる、ということに注意されたい。

仮引数に対する明示的なマッピング指示文を、ローカル手続内で指定してもよい。そのような指示文は、グローバル配列に対して適用されて、その各プロセッサ上にあるローカルな部分が、実引数又は関数結果として渡されている、とみなされる。そのような指示文が指定された場合、対応するマッピング指示文が、グローバル HPF 呼出し側から参照可能でなければならない。このためには、呼出し側に引用仕様宣言を記述するか、又は外来種別が HPF_LOCAL のモジュール内にローカル手続を記述し、ローカル手続を呼び出すグローバル HPF プログラム単位から、それを引用すればよい。

EXTRINSIC('HPF','LOCAL') 手続は、FORALL 構文の本体や INDEPENDENT ループの本体から、直接的にも間接的にも起動してはならない。ローカル手続は、複数の ENTRY をもっていてもよい。グローバル HPF 呼出し側には、HPF プログラムから起動できる各入口に対して、別々の外来引用仕様を記述しなければならない。

11.3.2 引数の結合

EXTRINSIC('HPF','LOCAL') 手続の仮引数がスカラの場合、ローカル手続の対応する仮引数は、同じ型及び型パラメタのスカラでなければならない。スカラは、組込み型、又は明示的にマップされていない構造型の場合だけ、グローバルプログラム単位から HPF_LOCAL 手続への引数に指定できる。外来手続が起動された時、ローカル手続にはスカラのローカルコピーが渡される。このコピーは、正当な HPF のスカラである。

EXTRINSIC('HPF','LOCAL') 手続の仮引数が配列の場合、ローカル手続内で宣言された仮引数は、同じ次元数、型、及び型パラメタの配列でなければならない。

外来引用仕様内の配列が順序的の場合、対応する実引数は、スカラ引数が渡される場合と同様に、全てのプロセッサ上に複製されて渡される。各ローカル仮引数は、実配列引数全体のコピーと結合する。外来引用仕様内の仮引数と、ローカル手続の宣言内の対応する仮引数は、同じ形状の形状明示配列であってよい。EXTRINSIC('HPF','LOCAL') 手続へ、複製されて渡される全ての順序的な仮引数は、INTENT(IN) 引数であるか、又は全プロセッサ上で、同じ値に更新されなければならない。

仮引数が非順序的な配列の場合、ローカル仮引数は、外来手続の起動時に、グローバル配列内のローカルに割り付けられた部分格子であるローカル配列と結合する。このローカル配列は、正当な HPF の配列である。

EXTRINSIC('HPF','LOCAL') 手続が関数の場合、ローカル手続は、HPF の外来関数と同じ型、型パラメタのスカラを返す関数である。ローカルに起動される各関数の返値は、同一でなければならない。

各物理プロセッサは、各 HPF 変数の、高々1 つのコピーをもつ。

次のような外来引用仕様を考えてみよう。

```
INTERFACE
  EXTRINSIC('HPF','LOCAL') FUNCTION MATZOH(X, Y) RESULT(Z)
    REAL, DIMENSION(:,:) :: X
    REAL, DIMENSION(:) :: Y
    REAL Z
    !HPF$ ALIGN WITH X(:,*) :: Y(:)
      ! この宣言は、size(Y) = size(X,1) であることを主張している。
    !HPF$ DISTRIBUTE X(BLOCK, CYCLIC)
  END FUNCTION
END INTERFACE
```

対応するローカル HPF 手続は、以下のように定義されているとする。

```
EXTRINSIC('HPF','LOCAL') FUNCTION MATZOH(XX, YY) RESULT(ZZ)
  REAL, DIMENSION(:,:) :: XX
  REAL, DIMENSION(5:) :: YY ! 下限が 5 の形状引継ぎ配列
  REAL ZZ
  NX1 = SIZE(XX, 1)
  LX1 = LBOUND(XX, 1)
  UX1 = UBOUND(XX, 1)
  NX2 = SIZE(XX, 2)
  LX2 = LBOUND(XX, 2)
  UX2 = UBOUND(XX, 2)
  NY  = SIZE(YY, 1)
  LY  = LBOUND(YY, 1)
  UY  = UBOUND(YY, 1)
  ...
END FUNCTION
```

2×2 のプロセッサ構成を用いた 4 プロセッサマシン上で (物理プロセッサ 1 つ当たり、1 つの抽象プロセッサが対応していると仮定する)、形状が 3×3 の (グローバル) 実配列引数 X、及び長さ 3 の実ベクトル引数 Y と共に、関数が起動されると仮定する。

すると、ローカルに起動された関数 MATZOH の各々は、以下のような実引数を受け取る。

Processor (1,1)	Processor (1,2)
X(1,1) X(1,3)	X(1,2)
X(2,1) X(2,3)	X(2,2)
Y(1)	Y(1)
Y(2)	Y(2)

Processor (2,1)	Processor (2,2)
X(3,1) X(3,3)	X(3,2)
Y(3)	Y(3)

各プロセッサが、 NX1、LX1、 UX1、 NX2、 LX2、 UX2、 NY、LY、及び UY に設定する値は、以下の通りである。

Processor (1,1)	Processor (1,2)
NX1 = 2 LX1 = 1 UX1 = 2	NX1 = 2 LX1 = 1 UX1 = 2
NX2 = 2 LX2 = 1 UX2 = 2	NX2 = 1 LX2 = 1 UX2 = 1
NY = 2 LY = 5 UY = 6	NY = 2 LY = 5 UY = 6

Processor (2,1)	Processor (2,2)
NX1 = 1 LX1 = 1 UX1 = 1	NX1 = 1 LX1 = 1 UX1 = 1
NX2 = 2 LX2 = 1 UX2 = 2	NX2 = 1 LX2 = 1 UX2 = 1
NY = 1 LY = 5 UY = 5	NY = 1 LY = 5 UY = 5

外来手続への実引数は、ポインタであってもよい。対応する仮引数には、POINTER 属性を指定できないので、仮引数は、グローバル HPF 内のポインタ指示先と結合することになる。ローカルなポインタを、グローバル HPF 内の指示先と結合させることはできない。したがって、実引数は、ポインタ成分を含む構造型であってはならない。

【仕様の根拠】 グローバルなポインタ変数は、少なくとも分散メモリマシン上では、ローカルなポインタ変数とは違う内部表現をもっていると考えられる。というのは、グローバルなアドレスの割り振りのために、付加的な情報をもつ必要があるからである。この制限は、将来取り除かれる可能性がある。【以上】

ALLOCATED や PRESENT のような問合せ組込み手続も、仕様通りの動作をすべきである。グローバル配列がローカル手続へ渡される時、受け取る配列要素の集合が空であるプロセッサもあり得る、ということに注意されたい。

11.3.3 HPF_SERIAL に関する特記事項

ここでは、HPF_SERIAL 副プログラムに適用される制約を述べる。

specification-directive、 *realign-directive*、又は *redistribute-directive* を、HPF_SERIAL 副プログラムやその引用仕様本体に指定してはならない。

【仕様の根拠】 HPF のマッピング指示文は、HPF_SERIAL 副プログラム内では意味をもたない。しかし、*independent-directive* を、HPF_SERIAL 副プログラム内で指定しても

よい、ということに注意されたい。というのは、 それにより、DO ループ、FORALL 文、又は FORALL 構文に関する、コンパイラにとって重要な情報が提供され得るからである。【以上】

HPF_SERIAL 手続の任意の仮データ実体と関数結果変数は、順序的だとみなされる。

HPF_SERIAL 副プログラム内では、HPF 又は HPF_LOCAL プログラム単位で宣言された共通ブロックと同じ名前の共通ブロックを宣言してはならない。さらに、HPF 又は HPF_LOCAL プログラム単位内に無名共通ブロックの宣言がある場合、HPF_SERIAL 副プログラム内では、無名共通ブロックを宣言してはならない。

グローバル HPF から引用される HPF_SERIAL 手続の仮引数や関数結果変数には、POINTER 属性を指定してはならない。グローバル HPF から引用される HPF_SERIAL 手続の仮引数や関数結果の部分実体にも、POINTER 属性を指定してはならない。

グローバル HPF から引用される HPF_SERIAL 手続の仮引数やその部分実体には、TARGET 属性を指定してはならない。

HPF_SERIAL 手続の仮手続引数は、HPF_SERIAL 手続でなければならない。

```
      PROGRAM MY_TEST
        INTERFACE
          EXTRINSIC('HPF','SERIAL') SUBROUTINE GRAPH_DISPLAY(DATA)
            INTEGER, INTENT(IN) :: DATA(:, :)
          END SUBROUTINE GRAPH_DISPLAY
        END INTERFACE

        INTEGER, PARAMETER :: X_SIZE = 1024, Y_SIZE = 1024

        INTEGER DATA_ARRAY(X_SIZE, Y_SIZE)
!HPF$   DISTRIBUTE DATA_ARRAY(BLOCK, BLOCK)

!  DATA_ARRAY を計算する
        ...
        CALL DISPLAY_DATA(DATA_ARRAY)
      END PROGRAM MY_TEST

! 画像表示用副プログラムの定義
! この手続は、なんらかの実装依存の方式で、
! DATA 中のデータのグラフをプロットする。

      EXTRINSIC('HPF','SERIAL') SUBROUTINE GRAPH_DISPLAY(DATA)
        INTEGER, INTENT(IN) :: DATA(:, :)
        INTEGER :: X_IDX, Y_IDX

        DO Y_IDX = LBOUND(DATA, 2), UBOUND(DATA, 2)
          DO X_IDX = LBOUND(DATA, 1), UBOUND(DATA, 1)
```

```
      ...
      END DO
    END DO
  END SUBROUTINE GRAPH_DISPLAY
```

11.4 C 言語との結合仕様

Fortran のユーザが直面する共通の問題は、他の言語、特に C や C プロトタイプで記述できる引用仕様をもった言語の手続を、呼び出す必要に迫られる、ということだろう。多くの Fortran の実装では、この問題を解決する方法が提供されているが、これらの方法には、ほとんどの場合移植性がない。

この節では、相互引用性に対する一般的な障害のほとんどを解消する一方で、移植性も維持できるような、C 言語で定義された手続に対する引用仕様の指定方法を述べる。

11.4.1 C 言語で定義された手続に対する引用仕様

手続が C 言語で定義されている、ということを、ユーザが指定したい場合には、第 6 章で規定されているように、`LANGUAGE = 'C'` という *extrinsic-spec-arg* 、又は `C` という *extrinsic-kind-keyword* を指定すればよい。

`EXTRINSIC (LANGUAGE = 'C')` が指定された C 言語の副プログラムに対しては、*attr-spec-extended*(H1102) の構文に関する制約が、以下のように拡張される。

制約: `LANGUAGE = 'C'` 関数の結果は、整数型、実数型、又は倍精度実数型のスカラでなければならない。

制約: `LANGUAGE = 'C'` 手続の仮引数は、形状引継ぎ配列であってはならない。また、`LANGUAGE = 'C'` 手続の仮引数に、`POINTER` 属性や `TARGET` 属性を指定したり、その部分実体に `POINTER` 属性を指定してはならない。

制約: `LANGUAGE = 'C'` 手続の仮引数の上下限は、定数宣言式でない宣言式であってはならない。また、仮引数の文字長パラメタは、定数宣言式でない宣言式であってはならない。

制約: `LANGUAGE = 'C'` サブルーチン の *dummy-arg-list* 中の *dummy-arg* として、*や仮手続を指定してはならない。

`EXTERNAL_NAME` 指定子中の *scalar-char-initialization-expr* の値により、C 言語における手続の名前が指定される。この値は、*function-stmt* や *subroutine-stmt* で宣言された手続名と同じである必要はない。`EXTERNAL_NAME` が省略された場合、手続名を小文字で記述した値が指定されているものとみなされる。

> 【利用者への助言】 `EXTERNAL_NAME` 指定子には、必ずしも、リンカが手続を認識する際の名前を指定する訳ではない、ということに注意されたい。そこに指定するのは、C 言語のプログラムから引用された場合に、手続が認識される名前であり、HPF コンパイラが、C コンパイラと同様の変換を行わなければならない名前である。

ユーザが、EXTERNAL_NAME 指定子に指定する名前は、例えば、最初の文字が下線の名前、大文字と小文字を区別しなければならない名前、といった HPF コンパイラに対しては許されない名前でもよい。

【以上】

LANGUAGE = 'C'という *extrinsic-spec-arg* により、コンパイラは、手続が C で定義されていることを認識できるので、C コンパイラが要求する方法による手続の起動を保証するための、適切な手順を踏むことができる。

【実装者への助言】 ベンダは、システムで利用可能な様々な C コンパイラが、様々な手続呼出し規約やデータ型の大きさを提供している場合、2 つ以上の C コンパイラをサポートしなければならない、と考えるかもしれない。ベンダは、例えば、LANGUAGE=指定子に、GNU_C という値を提供してもよいし、コンパイラへのオプションによりサポートしてもよい。【以上】

11.4.2 C 言語に対するデータ型変換の仕様

MAP_TO 属性、LAYOUT 属性、及び PASS_BY 属性からなる外来手続の仮引数の特性は、Fortran プログラムから C で定義された手続を引用できるようにするための主要な機能である。また、これらの属性により、ユーザは、引用された手続の実引数と仮引数とを結合するために必要な変換を指定できる。MAP_TO 属性により、コンパイラが、HPF のデータを C 手続のどのデータ型に変換するのかを指定できる。また、PASS_BY 属性により、仮引数への C ポインタが渡される必要があるか否かを指定できる。また、LAYOUT 属性により、配列を、Fortran 言語の配列要素順序から C 言語のそれに変換する必要があるか否かを指定できる。

C 言語に対しては、*attr-spec-extended*、*map-to-spec*、 *layout-spec*、及び *pass-by-spec* (H1102–H1105) に関する制約が、以下のように拡張される。

制約: MAP_TO 属性は、LANGUAGE = 'C'手続に対する明示的引用仕様内の、全ての仮引数と関数結果変数に指定しなければならない。

制約: 仮引数に対して指定する *map-to-spec* は、仮引数の型と適合していなければならない。(以下で述べる適合規則を参照せよ)

制約: LAYOUT 属性は、仮引数が配列の場合だけ指定できる。

制約: LAYOUT 属性を、大きさ引継ぎ配列に指定してはならない。

コンパイラが大文字と小文字を区別できる場合でも、*map-to-spec*、*layout-spec*、及び *pass-by-spec* の値に対しては、その区別は無い。コンパイラは、*map-to-spec*、*layout-spec*、及び *pass-by-spec* の値の識別の際、それらの中の空白を全て無視する。

実装者は、ISO_C というモジュールを提供して、その中で、C_VOID_POINTER という構造型を定義しなければならない。C_VOID_POINTER 型の成分は、非公開である。

【利用者への助言】 C_VOID_POINTER 型により、プログラム内で、void *ポインタを使用できるようになる。しかし、ユーザには、プログラムの Fortran 部分でその様なポイ

ンタを操作するいかなる方法も与えられない。というのは、非公開成分をもつ実体に対して、その型を定義しているモジュール外で I/O を行うことはできないし、その様な構造型の成分や構造体構成子を参照することもできないからだ。【以上】

LANGUAGE = 'C'に対する *map-to-spec* の値として許されているものは、'INT'、'LONG'、'SHORT'、'SIGNED_CHAR'、'FLOAT'、'DOUBLE'、'LONG_DOUBLE'、'CHAR'、'CHAR_PTR'、'VOID_PTR'、又はこれらの値を任意にカンマで区切って並べて、括弧で括ったものである。これらと適合する HPF の型は、以下の表に示されている。

値の並びが括弧で括られた *map-to-spec* は、並び中の各値が構造型の対応する成分と適合する場合、構造型仮引数と適合する。

PASS_BY 属性が指定された場合、*pass-by-spec* として許される値は、'VAL'、'*'、又は'**'である。PASS_BY 属性が省略された場合、PASS_BY ('VAL') が仮定される。*pass-by-spec* として、VAL という値が指定された場合、仮引数には、 INTENT(OUT) 属性も INTENT(INOUT) 属性も指定してはならない。*pass-by-spec* として、'*'又は'**'という値が指定された場合、結合する実引数は、変数でなければならない。

値が C である LANGUAGE=指定子が指定された手続の引用仕様本体内で、仮引数に対して指定する *map-to-spec* の値は、変換可能な型の少なくとも 1 つが、手続の C 言語による定義における、対応する仮引数の (C 言語の) データ型と同じでなければならない (あるいは、変換可能な型の 1 つと適合する型でなければならない)。手続の C 言語による定義における、関数の (C 言語の) データ型は、値が C である LANGUAGE=指定子が指定された関数の引用仕様本体内で、関数結果変数に対して指定された変換可能な型の 1 つでなければならない (あるいは、変換可能な型と等価な型でなければならない)。サブルーチンに、値が C である LANGUAGE=指定子が指定されている場合、手続の C 言語による定義には、void というデータ型を指定しなければならない。

スカラ仮引数が、組込み型、又は構造型 C_VOID_POINTER である場合の変換可能な型は、以下の表に示されている。

MAP_TO	**適合する Fortran の型**	各 PASS_BY に対応する **C 言語の型**		
		'VAL'	'*'	'**'
'INT'	INTEGER	int	int*	int**
'LONG'	INTEGER	long	long*	long**
'SHORT'	INTEGER	short	short*	short**
'SIGNED_CHAR'	INTEGER	signed char	signed char*	signed char**
'FLOAT'	REAL	float	float*	float**
'DOUBLE'	REAL	double	double*	double**
'LONG_DOUBLE'	REAL	double	double*	double**
'CHAR'	CHARACTER(1)	char	char*	char**
'CHAR_PTR'	CHARACTER	char*	char**	char***
'VOID_PTR'	C_VOID_POINTER	void*	void**	void***

配列の変換可能な型は、各次元に対して、左角括弧 ([)、仮引数の対応次元の寸法、右角括弧 (]) が指定されている同じ型のスカラ変数の変換可能な型と等しい。LAYOUT 属性が省略されている場合、実引数の各次元に対して、仮引数の次元は、右から左へと対応する。一方、

LAYOUT 属性に C_ARRAY という値が指定されている場合、実引数の各次元に対して、仮引数の次元は、左から右へと対応する。

LANGUAGE = 'C'の *layout-spec* として許されている値は、C_ARRAY である。

構造型スカラ変数の変換可能な型は、対応する成分が、構造型成分の変換可能な型のうちの 1 つであるような構造型である。

仮引数の型と、仮引数の変換可能な型との間で、精度、表現方式、許される値の範囲、又は記憶列の不一致がある場合、コンパイラは、C という値の LANGUAGE=指定子が宣言された手続の引用中、変換可能な型の実体に対して C 言語処理系が想定している表現と、仮引数の表現が適合していることを保証しなければならない。手続から戻る時には、コンパイラは、変数である実引数の値が、宣言された型と種別に戻ることを保証しなければならない。

型と変換先の型との間で、許される値の範囲が異なっていて、実引数やその部分実体の値が、変換先の型で許される範囲にない場合、それと結合する仮引数やその部分実体の値は、不定となる。逆に、仮引数やその部分実体の値が、結合する実引数の値として許される範囲になく、実引数が変数である場合、実引数やその部分実体の値は、不定となる。

【利用者への助言】 これらの規則は、相互引用機能の移植性を保証するために定められた。しかし、大きな実体の場合、データ型に用いられる表現方式と、変換可能な型に用いられる表現方式が一致していない場合、大きなオーバヘッドが生じる可能性があることに注意すべきである。【以上】

【利用者への助言】 引用された手続内で値が更新されないにもかかわらず、実引数の値が変わってしまう場合があり得る。例えば、

```
PROGRAM P
  INTERFACE
    EXTRINSIC(LANGUAGE='C') SUBROUTINE C_SUB(R,I)
      REAL(KIND(1.0D0)), MAP_TO('FLOAT'), PASS_BY('*') :: R
      INTEGER, MAP_TO('INT'), PASS_BY('*') :: I
    END SUBROUTINE C_SUB
  END INTERFACE
  REAL(KIND(0.0D0)) RR

  RR = 1.0D0 + 1.0D-10
  I = 123456789
  PRINT *, RR
  CALL C_SUB(RR, I)
  PRINT *, RR
END PROGRAM P

void c_sub(float *r, int *i)
{
}
```

の結果は、*r の値が、c_sub 内で更新されないにもかかわらず、以下のようになる可能性がある。

```
1.00000000010000000
1.00000000000000000
```

同様に、I の値は、更新されないにもかかわらず、c_sub の引用後不定となる可能性がある。

基本実数型以外の任意の型の仮引数に対して、変換先の型に float を指定しないようにしたり、倍精度実数型以外の任意の型の仮引数に対して、 変換先の型に double を指定しないようにするのは、良い方法であるが、変換先の型に int、又は long を指定する必要がある実体に対して、適切な仮引数の型を選択することは、あまり簡単ではないかもしれない。【以上】

仮配列引数に対して、*layout-spec* が省略された場合、配列要素順序は、Fortran で規定されている順序と同じである。指定された *layout-spec* の値が C_ARRAY の場合、手続引用中の、その配列の配列要素順序は、転置されることになる。

11.4.2.1 データ型変換の例

与えられた Fortran プログラム内の引用仕様本体に対して、どのような種類の C 言語の手続定義が許されるのかを明らかにするために、幾つかの例が役に立つだろう。例えば、以下のような引用仕様本体は、

```
INTERFACE
  EXTRINSIC('C') SUBROUTINE C_SUB(I, R, DARR, STRUCT)
    INTEGER, MAP_TO('INT') :: I
    REAL, MAP_TO('FLOAT'), PASS_BY('*') :: R
    REAL(KIND(1.0D0)), MAP_TO('DOUBLE') :: DARR(10)
    TYPE DT
      SEQUENCE
      INTEGER :: I, J
    END TYPE DT
    TYPE(DT), MAP_TO('(INT, LONG)'), PASS_BY('*') :: STRUCT
  END SUBROUTINE C_SUB
END INTERFACE
```

以下のようなプロトタイプ宣言をもつ C 言語の手続に対応し得る。

```
void c_sub(int i, float r*, double darr[10], struct {int i, long j} *)
```

以下の例のような LAYOUT 属性が指定された場合、

```
PROGRAM P
  INTERFACE
    EXTRINSIC('C') SUBROUTINE C_SUB(A, B)
      INTEGER, MAP_TO('INT') :: A(2,2)
      INTEGER, MAP_TO('INT'), LAYOUT('C_ARRAY') :: B(2,2)
    END SUBROUTINE C_SUB
  END INTERFACE

  INTEGER :: AA(2,2), BB(2,2)
  CALL C_SUB(AA, BB)
END PROGRAM P

void c_sub(int a[2][2], b[2][2])
```

AA と a の要素、及び BB と b の要素の対応は、以下のようになる。

```
AA(1,1)   a[0][0]              BB(1,1)   b[0][0]
AA(2,1)   a[0][1]              BB(2,1)   b[1][0]
AA(1,2)   a[1][0]              BB(1,2)   b[0][1]
AA(2,2)   a[1][1]              BB(2,2)   b[1][1]
```

11.5 Fortran 言語との結合仕様

外来種別の宣言で指定された言語が Fortran の場合、結合仕様は基本的に HPF に対するものと同じである。というのは、HPF 規格は、Fortran 規格に基づいているからである。この場合に考慮すべき点が少しある。

- Fortran 構文のみを使用すべきである。非同期 I/O や HPF ライブラリのような機能はサポートされない場合がある。
- この目的で利用される Fortran 言語処理系は、HPF_LOCAL マッピング問合せライブラリ、及び HPF_LOCAL_LIBRARY モジュールの引用をサポートするよう拡張されていることが望ましい。
- Fortran 言語処理系で外来プログラム単位をコンパイルするつもりなら、それらは、HPF ソースコードと一緒のファイルではなく、別のファイルに記述すべきである。
- プログラマは、HPF 指示文は全て無視され得る、と考えるべきである。

11.6 Fortran 77 言語との結合仕様

Fortran 77 は、言語の引用仕様に関しては、現在も本質的に ANSI/ISO Fortran 規格の部分集合である。従って、HPF からの Fortran 手続呼出しに関連するほとんどの事項は、HPF から Fortran 77 外来手続を呼び出す際にも当てはまる。しかし、Fortran と Fortran 77

との間には、以下のように、2 つの重要な違いがあるため、HPF からの引用仕様である EXTRINSIC(LANGUAGE='F77') の規格が、特にローカルモデルの場合に複雑になっている。

- 引数の渡し方が通常は異なる。典型的な Fortran 77 の実装では、副プログラムへの引数をアドレス (参照渡し) により渡す。つまり、データ型、大きさ、マッピングといった、実引数に関するその他の情報は何も渡されない。これに対して、HPF の実装では、しばしば記述子経由で変数を渡し、副プログラム内でそれらの情報を参照できるようにしている。

- 配列要素の、記憶領域への配置方法に関して利用可能な情報が、Fortran 77 と HPF のプログラマの間では、かなり異なる。

  Fortran 77 の場合、配列の宣言により、配列要素と線形に並んだ記憶領域との対応が正確に規定される。HPF の場合、複数のプロセッサへの配列要素のマッピングを (例えば、DISTRIBUTE 指示文や ALIGN 指示文によって) 制御したり、(例えば、HPF_ALIGNMENT、HPF_DISTRIBUTION 、又は HPF_TEMPLATE によって) 問合せたりすることはできるが、任意のプロセッサにローカルな、連続した記憶領域内で、配列要素がどの様に配置されているかに関する情報は全く無い。Fortran 90 の場合ですら、例えば、形状引継ぎ配列の場合などは、Fortran 77 配列の記憶列結合と順序結合の規則に従っているとは限らない。

  実際のところ、HPF コンパイラが異なれば、ローカルなデータの配置方法は異なる可能性がある。例えば、定型パタンを 最適化する為に、境界要素を余分に割り付けているかもしれないし、全てのプロセッサ上で、部分格子の大きさを等しくするために、ダミー要素の埋め込みを行っているかもしれない。実際、HPF コンパイラは、特定の方法でローカルなデータを配置するように義務づけられては「いない」し、例えば、SMP の場合など、配列宣言イメージ通りの配置順序を選択する HPF コンパイラもあり得る。

11.6.1 F77_LOCAL に関する特記事項

EXTRINSIC(F77_LOCAL) 引用仕様は、 HPF_LOCAL 及び FORTRAN_LOCAL の外来引用仕様を、Fortran 77 プログラマの必要に合わせて拡張したものである。

この EXTRINSIC 種別においては、先に述べたような、外来副プログラム引用のための構文が用いられる。これは、より正確には、EXTRINSIC(LANGUAGE='F77',MODEL='LOCAL') 引用仕様と記述できる。グローバルプログラム単位とローカル手続との間の制御の移行に関する基本的な規約は、先に第 11.1 節で述べたものが適用される。

引数の渡し方とマッピング情報に関するオプションを付加することによって、そのような引用仕様を使用するさまざまな動機を与えると共に、これら 2 つの言語の、引数の渡し方とデータマッピングという点における違いが扱いやすくなる。このようなオプションは、LAYOUT 属性と PASS_BY 属性により提供される。

11.6.2 F77_LOCAL 手続への引数の渡し方

典型的な Fortran 77 の実装では、引数は参照渡しであり、通常、最初のデータ要素の記憶領域のベースアドレスが渡される。また、その様な引数の順序結合も仮定されている。このような事情により、マップされたデータ構造を、HPF から F77_LOCAL 手続に渡す暗黙の方法と

しては、ローカルな記憶領域において、引数が割り付けられている場所のベースアドレスを渡すのが最も実際的である。実引数と仮引数の順序結合を可能にするために、全てのプロセッサ上で、データ並びに対する並べ変えや圧縮、又はその両方が必要となる場合がある。この方法は、マップされたデータを EXTRINSIC(F77_LOCAL) 手続に渡す最も安全な方法なので、暗黙の方法であるべきだが、性能上のコストが極めて大きくなりがちである。

引数渡しの 2 つ目の選択肢は、グローバル HPF 手続からローカル F77 手続へ、仮引数の順序結合は保証せずに、マップされた配列データを、「そのまま」渡す方法であり、この場合、プロセッサにローカルな配列の要素列を圧縮したり並べ変えるためのデータの移動を、ローカルに行う必要がない。言い換えれば、他の HPF 手続に、同じ引数を渡す場合より多くのデータ移動は行わなくてよい。仮引数の順序結合に対する保証は、実引数のローカルな要素列を並べ変えたり圧縮したりしなくて済む性能上のメリットのために犠牲にされる。ローカル手続のプログラマには、グローバル HPF プログラムにおける処理系依存の配置順序が利用可能でなければならない。

3 つ目の選択肢は、 EXTRINSIC(F77_LOCAL) 手続呼出しでも、HPF_LOCAL 風のローカル手続のプログラミングを可能にするためのものであり、HPF の実装や Fortran 90 の形状引継ぎ配列で典型的に行われているように、記述子やハンドル経由で配列を渡す方法である。ローカル手続から、この仮引数の要素を直接参照してはならず、特別なユーティリティ手続に対してそれを渡して、ローカル又はグローバルなマッピング情報などを得ることだけができる。

以下の属性は、EXTRINSIC(F77_LOCAL) 手続へ引数を渡す際の、上記 3 つの選択肢をサポートするのに十分である。

- LAYOUT('F77_ARRAY') ローカルな連続した記憶領域内で、配列が FORTRAN 77 の順序結合をなすことを示す。

  多くのコンパイラは、例えば、定型パタンの最適化のために境界要素を余分に付け加えたり、全てのプロセッサが各配列に対して等しい大きさの記憶領域を割り付けるために、特定のプロセッサ上では、配列内にダミー要素をもたせたりしている。

  ローカルな並べ変えはどんなものでも、INTERFACE 宣言内の DISTRIBUTE 指示文、又は ALIGN 指示文によって指定できるグローバルなデータの再マッピングに加えて行われる。

  LAYOUT 属性が省略された場合は、LAYOUT('F77_ARRAY') が仮定される。

- LAYOUT('HPF_ARRAY') ローカルなデータの並べ変えなしで、グローバル HPF 手続の場合と全く同じように、配列引数が渡されることを意味する。

  このオプションは、ローカルな並べ変えのオーバヘッドをなくしたい場合か、グローバル HPF コンパイラによる (境界要素付加、プロセッサ間の等サイズ割り付けといった) 配置方法のある特性を、ローカル手続においても保存したい場合に必要である。このオプションを指定する場合、ローカル手続のプログラマは、グローバル HPF コンパイラが採用する処理系依存の方法で、ローカルなデータを参照しなければならない。

さらに、INTERFACE 宣言内の各引数には、PASS_BY 属性を指定でき、それにより以下のことを指定できる。即ち、Fortran 77 風のデータ参照を行うために参照渡しにするか、特別なハンドル (恐らく、HPF の変数を渡す際に用いられる記述子経由の引数渡し) により、グローバル HPF 呼出し側が特別なマッピング情報を渡して、それをローカル Fortran 77 手続内で利用できるようにするかである。

- PASS_BY('*') ローカル手続が、ローカルな仮引数を、参照渡しによる F77 の変数と同様に参照できることを意味する。

- PASS_BY('HPF_HANDLE') ローカル手続が受け取るのは、グローバルプログラム単位の記述子であり、特別な問合せ手続と共にそれを用いると、有効なマッピング情報を得ることができる、ということを意味する。

ここで、暗黙の仮引数の属性は、LAYOUT('F77_ARRAY')、及び PASS_BY('*') であるから、順序結合は保証され、引数はその記憶領域へのポインタ経由で渡される。

【実装者への助言】

優れた EXTRINSIC(F77_LOCAL) の実装では、引数の渡し方とデータの並べ変えに関するオプションの提供に加えて、形状引継ぎ配列として宣言するのではなく、HPF の場合と同様に、任意の大きさのローカルな部分格子を宣言して、それらを参照できるようにする、という問題が解決されているべきである。

外来手続呼出しによって起動される各ローカル手続内で、グローバルにマップされたデータのローカルな配置方法に対応することも、Fortran 90 の配列問合せ関数や、HPF ライブラリの問合せサブルーチン無しでは難しい。付録 G で提案されている Fortran 77 ライブラリ関数の引用仕様の様に、グローバルにもローカルにも呼出せる特別の問合せ関数を、EXTRINSIC(F77_LOCAL) 手続インタフェースに追加して、より広い範囲のグローバル HPF のデータマッピングを、柔軟かつ効率的に利用できる様にすることが望ましい。

【以上】

11.6.3 F77_LOCAL 手続のプログラミング例

11.6.3.1 LAYOUT('F77_ARRAY') と PASS_BY('*')

ここでは、既定値である LAYOUT('F77_ARRAY') 属性と PASS_BY('*') 属性を用いた F77_LOCAL 手続のプログラミング例と、LAYOUT('HPF_ARRAY') 属性を用いた、ローカル手続内での問合せ手続の利用例を述べる。

- 呼出し側 HPF プログラム単位

```
      PROGRAM EXAMPLE

! 配列データと、値をチェックするためのコピーの宣言
      INTEGER, PARAMETER :: NX = 100, NY = 100
      REAL, DIMENSION(NX,NY) :: X, Y
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X, Y

! 全体の和は、各プロセッサ上の部分和を
! 結合することにより計算できる。
```

```
      REAL PARTIAL_SUM(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE PARTIAL_SUM(BLOCK)

! ローカルな部分格子である引数は、各プロセッサ毎に、
! 2次元配列として宣言される。
      INTEGER, DIMENSION(NUMBER_OF_PROCESSORS(),2) ::
     & LB, UB, NUMBER
!HPF$ DISTRIBUTE(BLOCK,*) :: LB, UB, NUMBER

! 引用仕様宣言
      INTERFACE

        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL1
     &   ( LB1, UB1, LB2, UB2, X , X_DESC )
        INTEGER, DIMENSION(:) :: LB1, UB1, LB2, UB2
        REAL,DIMENSION(:,:),LAYOUT('HPF_ARRAY') :: X
        REAL,DIMENSION(:,:),LAYOUT('HPF_ARRAY'),     &
               PASS_BY('HPF_HANDLE')           :: X_DESC

!HPF$   DISTRIBUTE(BLOCK) :: LB1, UB1, LB2, UB2
!HPF$   DISTRIBUTE(BLOCK,BLOCK) :: X, X_DESC
        END

        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL2(N,X,R)
        INTEGER N(:)
        REAL X(:,:), R(:)
! 既定値:
!       LAYOUT('F77_ARRAY')      行優先の連続した記憶領域
!       PASS_BY('*')             参照渡し (ローカルなアドレス)
!HPF$   DISTRIBUTE N(BLOCK)
!HPF$   DISTRIBUTE X(BLOCK,BLOCK)
!HPF$   DISTRIBUTE R(BLOCK)
        END

      END INTERFACE

! グローバル HPF だけを用いて、結果を求める。

      ! 初期化
      FORALL (I=1:NX,J=1:NY) X(I,J) = I + (J-1) * NX

      ! 全体の和の計算と報告
```

```
      PRINT *, 'Global HPF result: ',SUM(X)

! ローカルサブルーチンを用いて、結果を求める。

      ! 初期化 ( 整列の刻み幅は、1 と仮定する )
      CALL HPF_SUBGRID_INFO( Y, IERR, LB=lb, UB=UB )
      IF (IERR.NE.0) STOP 'ERROR!'
      CALL LOCAL1( LB(:,1), UB(:,1), LB(:,2), UB(:,2), Y , Y )

      ! 全体の和の計算と報告
      NUMBER = UB - LB + 1
      CALL LOCAL2 ( NUMBER(:,1) * NUMBER(:,2) , Y , PARTIAL_SUM )
      PRINT *, 'F77_LOCAL result #1 : ',SUM(PARTIAL_SUM)

      END
```

- 呼ばれ側 FORTRAN 77 手続

```
      SUBROUTINE LOCAL1( LB1, UB1, LB2, UB2, X , DESCRX )

      REAL X ( LB1 : UB1 , LB2 : UB2 )
      INTEGER DESCRX ( * )

! 最初の次元の、グローバル手続内での寸法を得る。
! これは、接頭文字列'F77_'をもつ HPF_LOCAL 種別の問合せ関数である。
      CALL F77_GLOBAL_SIZE ( NX , DESCRX , 1 )

! 配列要素の初期化
      DO J = LB2, UB2
        DO I = LB1, UB1
          X(I,J) = I + (J-1) * NX
        END DO
      END DO

      END



      SUBROUTINE LOCAL2(N,X,R)
! ここでは、グローバル手続内での添字との対応は重要ではない
! 部分格子の総数だけが渡される。
      REAL X(N)

```

```
      R = 0.
      DO I = 1, N
        R = R + X(I)
      END DO

      END
```

11.6.3.2 LAYOUT('HPF_ARRAY') と PASS_BY('HPF_HANDLE')

この例では、上の例の初期化部分だけを行う。F77_LOCAL 手続内で、部分格子問合せ関数を利用するための、HPF 風記述子又はハンドルを渡すために、PASS_BY('HPF_HANDLE') を用いる例と共に、HPF のマップされた配列を、再マップすることなく渡すために、LAYOUT('HPF_ARRAY') 属性を利用する例を述べる。また、「埋め込み配列」に特有のデータ参照法についても例示する。

- 呼出し側 HPF プログラム単位

```
      PROGRAM EXAMPLE

      INTEGER, PARAMETER :: NX = 100, NY = 100
      REAL, DIMENSION(NX,NY) :: Y
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: Y

! ローカルな部分格子である引数は、プロセッサ毎に、
! 2 次元配列として宣言される。
      INTEGER, DIMENSION(NUMBER_OF_PROCESSORS(),2) ::
     & LB, UB, LB_EMBED, UB_EMBED
!HPF$ DISTRIBUTE(BLOCK,*) :: LB, UB, LB_EMBED, UB_EMBED


! 引用仕様宣言

      INTERFACE

        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL1(
     &   LB1, UB1, LB_EMBED1, UB_EMBED1,
     &   LB2, UB2, LB_EMBED2, UB_EMBED2, X, X_DESC )
        INTEGER, DIMENSION(:) ::
     &   LB1, UB1, LB_EMBED1, UB_EMBED1,
     &   LB2, UB2, LB_EMBED2, UB_EMBED2
!             既定値で、X は参照渡しとなる。
        REAL, DIMENSION(:,:), LAYOUT('HPF_ARRAY') :: X
!             X_DESC は、記述子又は「ハンドル」により渡される。
```

```
        REAL, DIMENSION(:,:), LAYOUT('HPF_ARRAY'),
     &                    PASS_BY('HPF_HANDLE') :: X_DESC
!HPF$ DISTRIBUTE(BLOCK) :: LB1, UB1, LB_EMBED1, UB_EMBED1
!HPF$ DISTRIBUTE(BLOCK) :: LB2, UB2, LB_EMBED2, UB_EMBED2
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X
        END

      END INTERFACE

! 初期化
! ( 整列の刻み幅を 1 とし、次元の置換はないと仮定する )

      CALL HPF_SUBGRID_INFO( Y, IERR,
     & LB=LB, LB_EMBED=LB_EMBED,
     & UB=UB, UB_EMBED=UB_EMBED)
      IF (IERR.NE.0) STOP 'ERROR!'

      CALL LOCAL1(
     & LB(:,1), UB(:,1), LB_EMBED(:,1), UB_EMBED(:,1),
     & LB(:,2), UB(:,2), LB_EMBED(:,2), UB_EMBED(:,2), Y, Y )

      END
```

- 呼ばれ側 Fortran 77 手続

```
      SUBROUTINE LOCAL1(
     & LB1, UB1, LB_EMBED1, UB_EMBED1,
     & LB2, UB2, LB_EMBED2, UB_EMBED2, X, X_DESC )

! 部分格子は、「埋め込まれた」形で渡されている。
      REAL X ( LB_EMBED1 : UB_EMBED1 , LB_EMBED2 : UB_EMBED2 )

! この引数は、問合せ関数への入力としてのみ利用される。
      INTEGER X_DESC

! 最初の次元の、グローバル手続内での寸法を得る。
! これは、接頭文字列'F77_'をもつ HPF_LOCAL 種別の問合せ手続である。
      CALL F77_GLOBAL_SIZE(NX,X_DESC,1)

! 配列要素の初期化
! ループの範囲は、実引数として渡される配列要素上のみである。
      DO J = LB2, UB2
```

```
      DO I = LB1, UB1
        X(I,J) = I + (J-1) * NX
      END DO
    END DO

    END
```

11.7 外来手続のためのライブラリ

以下では、外来副プログラムのプログラミングに有用な手続の、Fortran 言語による結合仕様を述べる。

11.7.1 ローカル HPF 手続のためのライブラリ

ローカル HPF 手続内では、任意の HPF 組込み手続、ライブラリ手続を引用することができる。

> 【実装者への助言】そのような手続への引数は、ローカル配列である。実装方法に依存して、ローカル HPF 手続から引用される組込み手続、ライブラリ手続の実際のコードは、グローバル HPF コードから引用された場合のコードと同じであってもよいし、異なっていてもよい。【以上】

さらに、ローカルライブラリ手続 GLOBAL_ALIGNMENT、GLOBAL_DISTRIBUTION、及び GLOBAL_TEMPLATE が提供されているので、外来関数に対する実引数のグローバルなマッピングを問合せることができる。その他のローカルライブラリ手続は、実引数の大きさ、形状、及び配列の上下限を問合せるために提供される。これらのライブラリ手続は、仮引数の名前を入力として、対応するグローバル HPF の実引数に関する情報を返す。これらは、グローバル HPF コードから直接起動されたローカル手続内でのみ引用することができる。もしモジュールの機能が利用可能であれば、それらは、HPF_LOCAL_LIBRARY というモジュール内で定義される。それらを引用するローカル手続には、以下の文

```
USE HPF_LOCAL_LIBRARY
```

あるいは、これに代わる適切な機能をもつ文を記述しなければならない。

HPF ローカル手続ライブラリは、各物理プロセッサを 0 から $n-1$ までの整数値で識別する。ここで、n は、グローバル HPF 組込み関数 NUMBER_OF_PROCESSORS が返す値である。プロセッサ識別子は、ABSTRACT_TO_PHYSICAL の返値であるが、それにより、HPF のプロセッサ構成である抽象プロセッサと、物理プロセッサとの間に、一対一対応が定められる。さらに、ローカルライブラリ関数 MY_PROCESSOR は、それを呼び出したプロセッサの識別子を返す。

ローカル HPF ライブラリ手続のある引数が、グローバル HPF の実引数と結合するローカル手続の仮引数でなければならないときにはどんな場合でも、その結合は推移的ではない。つまり、ローカル手続の仮引数は、グローバル HPF から引用された手続の仮引数でなければならず、他のローカル副プログラムから引用された手続の仮引数であってはならない。

11.7.1.1 ブロック単位での仮引数参照

グローバル HPF 配列の物理プロセッサへのマッピングによって、1 つ、あるいはそれ以上の「**ブロック**」が、各プロセッサ上へ割り当てられる。ここで、**ブロック**とは、添字が連続している要素の集合のことである。あるプロセッサにマップされているブロックの数は、そのプロセッサにマップされている、各次元の添字が連続しているブロックの数を掛け合わせたものである。例えば、1 次元配列 X が、CYCLIC(4) で分散されている場合、それは、4 要素からなるブロックをもつ。ただし、最後のブロックは、SIZE(X) mod 4 要素になる可能性もある。一方、X が、CYCLIC(4) で分散されているテンプレート又は配列に整列しており、整列の刻み幅が 1 でない場合 (例えば、!HPF$ ALIGN X(I) WITH T(3*I))、そのブロックに含まれる要素は、4 より少ない。整列の刻み幅が 3 で、そのテンプレートが、各ブロック 4 要素のブロック-サイクリック分散の場合には、X の各ブロックは、1 要素又は 2 要素からなる。整列の刻み幅が 5 の場合、X の全てのブロックは、ちょうど 1 要素からなる。というのは、配列要素が全く整列されていないテンプレートのブロックは、ブロック数を数える際には無視されるからである。

HPF_LOCAL 副プログラムにおいて、仮引数と結合するグローバル配列引数の一部分は、ブロック毎に参照することができる。3 つのローカルライブラリ手続 LOCAL_BLKCNT、LOCAL_LINDEX、及び LOCAL_UINDEX により、ローカルな記憶領域内の特定のブロックを容易に参照できる。以下の例で、この様な目的のためにこれらの手続を利用する方法が示されている。そこでは、ローカルなデータは、一度に 1 ブロックずつ初期化されている。

```
EXTRINSIC(HPF_LOCAL) SUBROUTINE NEWKI_DONT_HEBLOCK(X)
REAL X(:,:,:)
INTEGER BL(3)
INTEGER, ALLOCATABLE :: LIND1(:), LIND2(:), LIND3(:)
INTEGER, ALLOCATABLE :: UIND1(:), UIND2(:), UIND3(:)

BL = LOCAL_BLKCNT(X)

ALLOCATE (LIND1(BL(1)))
ALLOCATE (LIND2(BL(2)))
ALLOCATE (LIND3(BL(3)))

ALLOCATE (UIND1(BL(1)))
ALLOCATE (UIND2(BL(2)))
ALLOCATE (UIND3(BL(3)))

LIND1 = LOCAL_LINDEX(X, DIM = 1)
UIND1 = LOCAL_UINDEX(X, DIM = 1)

LIND2 = LOCAL_LINDEX(X, DIM = 2)
UIND2 = LOCAL_UINDEX(X, DIM = 2)
```

```
LIND3 = LOCAL_LINDEX(X, DIM = 3)
UIND3 = LOCAL_UINDEX(X, DIM = 3)

DO IB1 = 1, BL(1)
  DO IB2 = 1, BL(2)
    DO IB3 = 1, BL(3)
      FORALL (I1 = LIND1(IB1) : UIND1(IB1),  &
              I2 = LIND2(IB2) : UIND2(IB2),  &
              I3 = LIND3(IB3) : UIND3(IB3) ) &
                X(I1, I2, I3) = IB1 + 10*IB2 + 100*IB3
    ENDDO
  ENDDO
ENDDO
END SUBROUTINE NEWKI_DONT_HEBLOCK
```

GLOBAL_ALIGNMENT(ARRAY, ...)

HPF の問合せサブルーチン HPF_ALIGNMENT と引用仕様や動作は同じ。ただしローカル配列に関する情報ではなく、ローカル仮引数 ARRAY と結合するグローバル HPF の実配列引数に関する情報を返す。

GLOBAL_DISTRIBUTION(ARRAY, ...)

HPF の問合せサブルーチン HPF_DISTRIBUTION と引用仕様や動作は同じ。ただしローカル配列に関する情報ではなく、ローカル仮引数 ARRAY と結合するグローバル HPF の実配列引数に関する情報を返す。

GLOBAL_TEMPLATE(ARRAY, ...)

HPF の問合せサブルーチン HPF_TEMPLATE と引用仕様や動作は同じ。ただしローカル配列に関する情報ではなく、ローカル仮引数 ARRAY と結合するグローバル HPF の実配列引数に関する情報を返す。

GLOBAL_SHAPE(SOURCE)

機能. HPF_LOCAL 手続の配列又はスカラ仮引数と結合するグローバル HPF の実引数の形状を返す。

分類. 問合せ関数。

引数.

SOURCE　任意の型でよい。配列でもスカラでもよい。グローバル HPF の実引数と結合する HPF_LOCAL 手続の仮引数でなければならない。

結果の型、型パラメタ、及び形状. 結果は、基本整数型 1 次元配列であって、大きさが SOURCE の次元数と同じとする。

結果の値. 結果の値は、SOURCE と結合するグローバル実引数の形状とする。

例示. A が、 INTEGER A(3:100, 200) と宣言されていて、B と結合する引数であるとき、GLOBAL_SHAPE(B) の値は、(/98, 200/) になる。B が、部分配列 A(5:10, 10) と結合する引数のとき、GLOBAL_SHAPE(B) の値は、(/6/) になる。

GLOBAL_SIZE(ARRAY, DIM)

省略可能な引数. DIM

機能. HPF_LOCAL 手続の仮配列引数と結合するグローバル HPF の実配列引数の、指定された次元の寸法を返す。

分類. 問合せ関数

引数.

ARRAY 任意の型でよい。スカラであってはならない。グローバル HPF の実引数と結合する HPF_LOCAL 手続の仮引数でなければならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで、n は ARRAY の次元数とする。

結果の型、型パラメタ、及び形状. 基本整数型スカラとする。

結果の値. 結果の値は、ARRAY と結合する実引数の DIM 次元目の寸法と同じであるか、又は DIM が省略されたとき、ARRAY と結合する実引数の要素の総数と同じとする。

例示. A が、 INTEGER A(3:10, 10) と宣言されていて、B と結合する引数であるとき、GLOBAL_SIZE(B, 1) の値は、8 になる。B が、部分配列 A(5:10, 2:4) と結合する引数のとき、GLOBAL_SIZE(B) の値は、18 になる。

ABSTRACT_TO_PHYSICAL(ARRAY, INDEX, PROC)

機能. グローバル実配列引数に対して指定されている抽象プロセッサに対応する物理プロセッサの、プロセッサ識別子を返す。

分類. サブルーチン。

引数.

ARRAY 任意の型でよい。グローバル HPF の実配列引数と結合する仮配列でなければならない。INTENT(IN) 引数とする。

INDEX　整数型 1 次元配列であって、その値は、グローバル HPF において、配列がマップされている抽象プロセッサの座標でなければならない。INTENT(IN) 引数とする。INDEX の大きさは、抽象プロセッサの次元数と同一でなければならない。i 番目の要素は、1 から e_i までの範囲の値でなければならない。ここで、e_i は、抽象プロセッサの i 次元目の寸法とする。

PROC　整数型スカラでなければならない。INTENT(OUT) 引数とする。INDEX によって指定された抽象プロセッサに対応する物理プロセッサの識別子を受け取る。

PHYSICAL_TO_ABSTRACT(ARRAY, PROC, INDEX)

機能. 指定された物理プロセッサに対応する、グローバル実配列引数がマップされている抽象プロセッサの座標を返す。

分類. サブルーチン。

引数.

ARRAY　任意の型でよい。グローバル HPF の実配列引数と結合する仮配列でなければならない。INTENT(IN) 引数とする。

PROC　基本実数型スカラでなければならない。INTENT(IN) 引数とする。物理プロセッサの識別子を指定する。

INDEX　整数型 1 次元配列でなければならない。INTENT(OUT) 引数とする。INDEX の大きさは、グローバル HPF において配列がマップされている抽象プロセッサの次元数と同一でなければならない。INDEX は、PROC により指定された物理プロセッサに対応する抽象プロセッサの座標を受け取る。i 番目の要素は、1 から e_i までの範囲の値でなければならない。ここで、e_i は、抽象プロセッサの i 次元目の寸法とする。

この手続は、抽象プロセッサと物理プロセッサが 1 対 1 に対応するシステム上でだけ引用することができる。この対応が 1 対多であるようなシステムでは、等価な処理系依存の手続が提供される。

LOCAL_TO_GLOBAL(ARRAY, L_INDEX, G_INDEX)

機能. ローカル仮配列のローカルな座標を、それと結合するグローバル HPF の実配列引数の、対応するグローバルな座標に変換する。

分類. サブルーチン。

引数.

ARRAY　任意の型でよい。グローバル HPF の実配列引数と結合する仮配列でなければならない。INTENT(IN) 引数とする。

L_INDEX　整数型 1 次元配列であって、大きさが ARRAY の次元数と同一でなければならない。INTENT(IN) 引数とする。ローカル仮配列 ARRAY の要素の座標をもつ。i 番目の要素は、1 から e_i までの範囲の値でなければならない。ここで、e_i は、ARRAY の i 次元目の寸法とする。

G_INDEX　整数型 1 次元配列であって、大きさが ARRAY の次元数と同一でなければならない。INTENT(OUT) 引数とする。L_INDEX により指定されたローカル配列の要素に対応する、グローバル HPF の実配列引数の座標を受け取る。i 番目の要素は、1 から e_i までの範囲の値とする。ここで、e_i は、ARRAY と結合するグローバル HPF の実配列引数の i 次元目の寸法とする。

GLOBAL_TO_LOCAL(ARRAY, G_INDEX, L_INDEX, LOCAL, NCOPIES, PROCS)

省略可能な引数. L_INDEX, LOCAL, NCOPIES, PROCS

機能. グローバル HPF の実配列引数のグローバルな座標を、それと結合するローカル仮配列の、対応するローカルな座標に変換する。

分類. サブルーチン。

引数.

ARRAY　任意の型でよい。グローバル HPF の実配列引数と結合する仮配列でなければならない。INTENT(IN) 引数とする。

G_INDEX　整数型 1 次元配列であって、大きさが ARRAY の次元数と同一でなければならない。INTENT(IN) 引数とする。ローカル仮配列 ARRAY と結合する、グローバル HPF の実配列引数の要素の座標をもつ。i 番目の要素は、1 から e_i までの範囲の値でなければならない。ここで、e_i は、ARRAY と結合するグローバル HPF の実配列引数の i 次元目の寸法とする。

L_INDEX (省略可能)　整数型 1 次元配列であって、大きさが ARRAY の次元数と同一でなければならない。INTENT(OUT) 引数とする。G_INDEX によって指定された、グローバル実配列引数の要素に対応する、ローカル仮配列の座標を受け取る。(同一のグローバル配列要素のコピーを所有するプロセッサ上では、これらの座標は同一である。) i 番目の要素は、1 から e_i までの範囲の値とする。ここで、e_i は、ARRAY の i 次元目の寸法とする。

LOCAL (省略可能)　論理型スカラでなければならない。INTENT(OUT) 引数とする。ローカル配列がグローバル配列要素のコピーをもつとき、.TRUE. に、そうでないとき、.FALSE. に設定されるとする。

NCOPIES (省略可能)　整数型スカラでなければならない。INTENT(OUT) 引数とする。指定されたグローバル実配列の要素のコピーを所有するプロセッサ数が設定される。

PROCS (省略可能) 整数型 1 次元配列であって、大きさが、グローバル実配列の指定された要素のコピーを所有するプロセッサ数以上でなければならない。そのようなプロセッサの識別子が、PROCS に設定される。その出現順序は、処理系依存とする。

MY_PROCESSOR()

機能. 呼び出した物理プロセッサの識別子を返す。

分類. ピュア関数。

結果の型、型パラメタ、及び形状. 結果は、基本整数型スカラとする。

結果の値. 呼出しを行った物理プロセッサの識別子を返す。$0 \leq$ MY_PROCESSOR $\leq n-1$ の範囲の値とする。ここで、n は、NUMBER_OF_PROCESSORS が返す値とする。

LOCAL_BLKCNT(ARRAY, DIM, PROC)

省略可能な引数. DIM, PROC.

機能. 与えられたプロセッサ上に割り当てられている配列の、各次元の要素のブロック数を返すか、又は指定された次元における要素のブロック数を返す。

分類. ピュア関数。

引数.

ARRAY 任意の型でよい。グローバル HPF の実配列引数と結合する仮配列でなければならない。

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければななならない。ここで、n は、ARRAY の次元数とする。対応する実引数は、省略可能な仮引数であってはならない。

PROC (省略可能) 整数型スカラでなければならない。有効なプロセッサ番号でなければならない。

結果の型、型パラメタ、及び形状. 結果は、基本整数型とする。DIM が指定されたときスカラとする。そうでなければ、結果は 1 次元配列であって、大きさが n とする。ここで、n は、ARRAY の次元数とする。

結果の値.

場合 *(i)*: LOCAL_BLKCNT(ARRAY, DIM, PROC) の値は、プロセッサ PROC にマップされている ARRAY の最終整列先の、DIM 次元目のブロック数とする。ARRAY の要素が、少なくとも 1 つは整列されているブロックだけを数えるものとする。

場合 *(ii)*: LOCAL_BLKCNT(ARRAY, DIM) は、LOCAL_BLKCNT(ARRAY, DIM, PROC=MY_PROCESSOR()) と同じ値を返す。

場合 *(iii)*: LOCAL_BLKCNT(ARRAY) の値は、$i = 1, \ldots, n$ に対して、i 番目の成分が、LOCAL_BLKCNT(ARRAY, i) と同じとする。ここで、n は、ARRAY の次元数とする。

例示. 以下のように宣言されたとき、

```
      REAL A(20,20), B(10)
!HPF$ TEMPLATE T(100,100)
!HPF$ ALIGN B(J) WITH A(*,J)
!HPF$ ALIGN A(I,J) WITH T(3*I, 2*J)
!HPF$ PROCESSORS PR(5,5)
!HPF$ DISTRIBUTE T(CYCLIC(3), CYCLIC(3)) ONTO PR
!HPF$ CALL LOCAL_COMPUTE(A, B)
      ...
      ...
      ...
      EXTRINSIC(HPF_LOCAL) SUBROUTINE LOCAL_COMPUTE(X, Y)
      USE HPF_LOCAL_LIBRARY
      REAL X(:,:), Y(:)
      INTEGER NBY(1), NBX(2)
      NBX = LOCAL_BLKCNT(X)
      NBY = LOCAL_BLKCNT(Y)
```

PR(2,4) に対応する物理プロセッサにおいて、NBX の値は、(/4,3/) になり、NBY の値は、(/1/) になる。

LOCAL_LINDEX(ARRAY, DIM, PROC)

省略可能な引数. PROC.

機能. 仮配列引数の与えられた次元における、プロセッサ上の全てのブロックのローカルな添字の下限を返す。

分類. ピュア関数。

引数.

ARRAY　任意の型でよい。グローバル HPF の実配列引数と結合する仮配列でなければならない。

DIM　整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで、n は、ARRAY の次元数とする。

PROC (省略可能) 整数型スカラでなければならない。有効なプロセッサ番号でなければならない。

結果の型、型パラメタ、及び形状. 結果は、基本整数型 1 次元配列であって、大きさは b とする。

ここで、b は、LOCAL_BLKCNT(ARRAY, DIM [, PROC]) の返値とする。

結果の値.

場合 *(i):* LOCAL_LINDEX(ARRAY, DIM, PROC) の i 番目の成分の値は、プロセッサ PROC における、ARRAY の DIM 次元目の、i 番目のブロックの最初の要素のローカルな添字とする。i 番目の要素は、1 から e_i までの範囲の値とする。ここで、e_i は、ARRAY の i 次元目の寸法とする。

場合 *(ii):* LOCAL_LINDEX(ARRAY, DIM) は、LOCAL_LINDEX(ARRAY, DIM, PROC=MY_PROCESSOR()) と同じ値を返す。

例示. LOCAL_BLKCNT の例と同じ宣言があるとき、PR(2,4) に対応する物理プロセッサにおいて、

LOCAL_LINDEX(X, DIM=1) の値は、(/1,2,3,4/) になる。LOCAL_LINDEX(X, DIM=2) の値は、(/1,3,4/) になる。

LOCAL_UINDEX(ARRAY, DIM, PROC)

省略可能な引数. PROC.

機能. 仮配列引数の与えられた次元における、プロセッサ上の全てのブロックのローカルな添字の上限を返す。

分類. ピュア関数。

引数.

ARRAY 任意の型でよい。グローバル HPF の実配列引数と結合する仮配列でなければならない。

DIM 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで、n は、ARRAY の次元数とする。

PROC (省略可能) 整数型スカラでなければならない。有効なプロセッサ番号でなければならない。

結果の型、型パラメタ、及び形状. 結果は、基本整数型 1 次元配列であって、大きさは b とする。ここで、b は、LOCAL_BLKCNT(ARRAY, DIM [, PROC]) の返値とする。

結果の値.

場合 *(i)*: LOCAL_UINDEX(ARRAY, DIM, PROC) の i 番目の成分の値は、プロセッサ PROC における、ARRAY の DIM 次元目の、i 番目のブロックの最後の要素のローカルな添字とする。i 番目の要素は、1 から e_i までの範囲の値とする。ここで、e_i は、ARRAY の i 次元目の寸法とする。

場合 *(ii)*: LOCAL_UINDEX(ARRAY, DIM) は、LOCAL_UINDEX(ARRAY, DIM, PROC=MY_PROCESSOR()) と同じ値を返す。

例示. LOCAL_BLKCNT の例と同じ宣言があるとき、PR(2,4) に対応する物理プロセッサにおいて、

LOCAL_UINDEX(X, DIM=1) の値は、(/1,2,3,4/) になる。LOCAL_UINDEX(X, DIM=2) の値は、(/1,3,4/) になる。

11.7.2 シリアル外来種別からのライブラリ引用

SERIAL 副プログラムは、HPF_ALIGNMENT、HPF_DISTRIBUTION、 HPF_TEMPLATE を除く、任意の HPF_LIBRARY 手続や HPF 組込み関数を引用できる。SERIAL プログラム単位の有効域内で、HPF_LOCAL_LIBRARY モジュールを引用してはならない。

組込み関数 NUMBER_OF_PROCESSORS や PROCESSORS_SHAPE の返値は、グローバル HPF 内で引用された場合と同一である。

第12章　HPF組込み手続とライブラリ手続の公認拡張

本章では、HPF Version 2.0 に対する拡張として公認された組込み手続とライブラリ手続に関して述べる。

拡張組込み手続には、Fortran の TRANSPOSE 組込み関数を汎用化した転置関数が含まれる。アルゴリズムによっては、多次元配列を様々な次元軸に沿って参照しなければならないことがある。今日の計算機では通常、配列要素が局所メモリ上で連続するよう、内側ループで配列の 1 次元目をなぞるようにするのがもっともよい。汎用化された転置は、このようなデータの再編成に必要とされるもので、単なるデータの再配置とは異なる。多くの場合、転置の結果は、第 1 次元が *dist-format*(*) で分散された変数に代入される。

次元数、及び入れ換えられる次元軸の集合が既知ならば、この種の操作には FORALL が適している。たとえば以下のようになる。

```
FORALL(I1 = 1:SIZE(ARRAY,1))
   FORALL(I2 = 1:SIZE(ARRAY,2))
      FORALL(I3 = 1:SIZE(ARRAY,3))
         RESULT(I3,I1,I2) = ARRAY(I1,I2,I3)
      ENDFORALL
   ENDFORALL
ENDFORALL
```

しかし、入力の次元軸と結果の次元軸との関係が可変であるときには、FORALL による表現は不自由である。そこで、TRANSPOSE 組込み関数は汎用化され、次元数が 0 でない任意の入力配列 (転置される) と、入力配列の次元数と同じ大きさを持つ整数型 1 次元配列 (次元軸の順列を与える) が引数に指定できるようになった。

引数の既定値により、この関数は既存の Fortran 関数である 1 引数の TRANSPOSE への拡張となる。

新しい 2 つの組込み問合せ関数 ACTIVE_NUM_PROCS と ACTIVE_PROCS_SHAPE は、プログラムを実行する部分プロセッサの大きさと形状を知るために利用できる。それらは ON 構文で変更される。

拡張ライブラリは、マッピング問合せサブルーチン群から成る。HPF_ALIGNMENT と HPF_TEMPLATE の拡張版では、新たに省略可能な出力引数 DYNAMIC を指定できる。これによりプログラムは、実体やその最終整列先が DYNAMIC 属性を持つかどうか知ることができる。HPF_DISTRIBUTION の改良版と新しい 2 つのマッピング問合せサブルーチンは、GEN_BLOCK 分散形式と INDIRECT 分散形式のマッピングを知るために特に役立つ。

ACTIVE_NUM_PROCS(DIM)

省略可能な引数. DIM

機能. 最内側の ON ブロックで決まる、現在プログラムを実行中のプロセッサの総数、又はプロセッサ配列の指定された次元における、現在プログラムを実行中のプロセッサ数を返す。

分類. プロセッサ問合せ関数

引数.

DIM (省略可能) 整数型スカラであって、$1 \leq$ DIM $\leq n$ の範囲の値でなければならない。ここで n は、プロセッサ配列の次元数とする。

結果の型、型パラメタ 及び 形状. 基本整数型スカラとする。

結果の値. 結果の値は、囲まれる最内側の ON ブロックで決まるプロセッサ配列の第 DIM 次元の寸法とする。DIM が省略されたときは、そのようなプロセッサ配列の要素の総数とする。結果は常に 0 より大きいものとする。囲まれる ON ブロックがないとき、結果の値は NUMBER_OF_PROCESSORS() の結果と同じとする。

例示. 以下のプログラム断片

```
INTEGER X(16, 3)
!hpf$ TEMPLATE T(16, 8)
!hpf$ PROCESSORS PROCS(4, 4)
!hpf$ ALIGN X(I, J) WITH T(I, 3*J-1)
!hpf$ DISTRIBUTE T(CYCLIC(2), BLOCK) ONTO PROCS
!hpf$ ON     (PROCS(:,:)) BEGIN
!hpf$ ON HOME(X(2:12:10, :)) BEGIN
PRINT *, ACTIVE_NUM_PROCS()
PRINT *, ACTIVE_NUM_PROCS(DIM=1)
PRINT *, ACTIVE_NUM_PROCS(DIM=2)
!hpf$ END ON
!hpf$ END ON
```

は、プログラムを実行するハードウェアプロセッサ配列の大きさや形状とは無関係に、6、2、3 を印字する。

ACTIVE_PROCS_SHAPE()

機能. 最内側の ON ブロックで決まる、現在の活動プロセッサ配列の形状を返す。

分類. プロセッサ問合せ関数

引数. なし

結果の型、型パラメタ 及び 形状. 基本整数型 1 次元配列で、囲まれる最内側の ON ブロックで決まるプロセッサ配列の次元数に等しい大きさとする。

結果の値. 結果の値は、囲まれる最内側の ON ブロックで決まるプロセッサ配列の形状とする。囲まれる ONブロックがないとき、結果の値は PROCESSORS_SHAPE() の結果と同じとする。

例示. 以下のプログラム断片

```
INTEGER X(16, 3)
!hpf$ TEMPLATE T(16, 8)
!hpf$ PROCESSORS PROCS(4, 4)
!hpf$ ALIGN X(I, J) WITH T(I, 3*J-1)
!hpf$ DISTRIBUTE T(CYCLIC(2), BLOCK) ONTO PROCS
!hpf$ ON     (PROCS(:,:)) BEGIN
    PRINT *, ACTIVE_PROCS_SHAPE()
    !hpf$ ON HOME(X(2:12:10, :)) BEGIN
    PRINT *, ACTIVE_PROCS_SHAPE()
    !hpf$ END ON
!hpf$ END ON
```

は、プログラムを実行するハードウェアプロセッサ配列の大きさや形状とは無関係に、$4, 4$ と $2, 3$ を印字する。

TRANSPOSE(ARRAY,ORDER)

省略可能な引数. ORDER

機能. 配列の次元軸を入れ換える (汎用の転置)。

分類. 変型関数

引数.

ARRAY　どの型でもよい。配列でなければならない。

ORDER (省略可能) 整数型 1 次元配列でなければならない。ARRAY の次元数と等しい大きさを持たなければならない。その要素は、$(1, 2, \ldots, n)$ の順列でなければならない。ここで n は、RANK(ARRAY) とする。

結果の型、型パラメタ 及び 形状. ARRAY と同じ次元数、型、型パラメタの配列とする。形状は、RS(ORDER) == AS という関係を満たす。ここで、RS は結果の形状とし、AS は SHAPE(ARRAY) とする。ORDER が省略されたとき、既定値は $(n, n-1, \ldots, 1)$ とする。ここで n は、RANK(ARRAY) とする。

結果の値. 結果の要素 $(j_1, j_2, \ldots, j_n)$ は、ARRAY$(j_{order(1)}, j_{order(2)}, \ldots, j_{order(n)})$ とする。

例示. 2 次元配列に対する TRANSPOSE(ARRAY) は、通常の行列の転置になる。

ARRAY の形状が $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ で、ARRAY(1,:,:) が、$\begin{bmatrix} 111 & 112 & 113 \\ 121 & 122 & 123 \end{bmatrix}$ で、ORDER が $\begin{bmatrix} 3 & 1 & 2 \end{bmatrix}$ のとき、結果の形状は、$\begin{bmatrix} 2 & 3 & 1 \end{bmatrix}$ になる。R を結果とすると、R(:,:,1) は ARRAY(1,:,:) と同じになる。ARRAY の第 i 次元が、結果の第 ORDER(i) 次元となる。

12.2 拡張ライブラリ手続きの仕様

HPF_ALIGNMENT(ALIGNEE, LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES)

省略可能な引数. LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES

機能. 変数と、それが最終的に整列する *align-target* (配列またはテンプレート) との対応に関する情報を返す。

分類. マッピング問合せサブルーチン

引数.

ALIGNEE どの型でもよく、スカラでも配列でもよい。大きさ引継ぎ配列であってはならない。空状態のポインタ又は割り付けられていない割付け配列であってはならない。INTENT (IN) 引数とする。

ALIGNEE がポインタであるときは、指示先の整列に関する情報が返されるものとする。指示先は大きさ引継ぎの仮引数又は大きさ引継ぎの仮引数の部分配列であってはならない。

LB (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。ALIGNEE の i 番目の次元軸の最初の要素が、対応する *align-target* の次元軸の LB(i) 番目の要素に最終的に整列している。ALIGNEE の i 番目の次元軸が縮退しているとき、LB(i) に設定される値は実装依存とする。

UB (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。ALIGNEE の i 番目の次元軸の最後の要素が、対応する *align-target* の次元軸の UB(i) 番目の要素に最終的に整列している。ALIGNEE の i 番目の次元軸が縮退しているとき、UB(i) に設定される値は実装依存とする。

STRIDE (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。STRIDE の i 番目の要素には、ALIGNEE の i 番目の次元軸に沿って各要素が整列するときの刻み幅が設定される。ALIGNEE の i 番目の次元軸が縮退しているとき、STRIDE(i) は 0 に設定される。

AXIS_MAP (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。AXIS_MAP の i 番目の要素には、ALIGNEE の i 番目の次元軸に対応する *align-target* の次元軸が設定される。ALIGNEE の i 番目の次元軸が縮退しているとき、AXIS_MAP(i) は 0 に設定される。

IDENTITY_MAP (省略可能) 基本論理型スカラでなければならない。INTENT (OUT) 引数とする。ALIGNEE に対応する最終 *align-target* が ALIGNEE と同じ形状を持ち、次元軸が恒等置換を使ってマップされ、刻み幅がすべて正である (したがって形状の制限から 1 に等しい) とき、真が設定される。それ以外のとき、偽が設定される。変数が、ALIGN 指示文や REALIGN 指示文の *alignee* に現れず、INHERIT 属性も持たないとき、IDENTITY_MAP は真に設定される。ただし、その他の状況でも真となることがある。

DYNAMIC (省略可能) 基本論理型スカラでなければならない。INTENT (OUT) 引数とする。ALIGNEE が DYNAMIC 属性を持つとき、真が設定される。それ以外のとき、偽が設定される。

NCOPIES (省略可能) 基本整数型スカラでなければならない。INTENT (OUT) 引数とする。*align-target* に最終的に整列している ALIGNEE の複製の数が設定される。複製がないときは 1 が設定される。

ALIGNEE がスカラのとき、LB、UB、STRIDE、AXIS_MAP には何も設定されない。

例示. 以下の宣言が与えられ、

```
      REAL PI = 3.1415927
      DIMENSION A(10,10),B(20,30),C(20,40,10),D(40)
!HPF$ TEMPLATE T(40,20)
!HPF$ DYNAMIC A
!HPF$ ALIGN A(I,:) WITH T(1+3*I,2:20:2)
!HPF$ ALIGN C(I,*,J) WITH T(J,21-I)
!HPF$ ALIGN D(I) WITH T(I,4)
!HPF$ PROCESSORS PROCS(4,2), SCALARPROC
!HPF$ DISTRIBUTE T(BLOCK,BLOCK) ONTO PROCS
!HPF$ DISTRIBUTE B(CYCLIC,BLOCK) ONTO PROCS
!HPF$ DISTRIBUTE ONTO SCALARPROC :: PI
```

実際のマッピングが宣言通りに行なわれたとすると、HPF_ALIGNMENT の呼出しの結果は以下のようになる。

	A	B	C	D
LB	[4, 2]	[1, 1]	[20, N/A, 1]	[1]
UB	[31, 20]	[20, 30]	[1, N/A, 10]	[40]
STRIDE	[3, 2]	[1, 1]	[-1, 0, 1]	[1]
AXIS_MAP	[1, 2]	[1, 2]	[2, 0, 1]	[1]
IDENTITY_MAP	偽	真	偽	偽
DYNAMIC	真	偽	偽	偽
NCOPIES	1	1	1	1

ここで、「N/A」は実装依存の結果を表わす。NCOPIES の利用法を次の例で見る。

```
      LOGICAL BOZO(20,20),RONALD_MCDONALD(20)
!HPF$ TEMPLATE EMMETT_KELLY(100,100)
!HPF$ ALIGN RONALD_MCDONALD(I) WITH BOZO(I,*)
!HPF$ ALIGN BOZO(J,K) WITH EMMETT_KELLY(J,5*K)
```

このとき、CALL HPF_ALIGNMENT(RONALD_MCDONALD, NCOPIES = NC) は NC に 20 を設定する。また次の例では、

```
      LOGICAL BOZO(20,20),RONALD_MCDONALD(20)
!HPF$ TEMPLATE WILLIE_WHISTLE(100)
!HPF$ ALIGN RONALD_MCDONALD(I) WITH BOZO(I,*)
!HPF$ ALIGN BOZO(J,*) WITH WILLIE_WHISTLE(5*J)
```

このとき、CALL HPF_ALIGNMENT(RONALD_MCDONALD, NCOPIES = NC) は NC に 1 を設定する。

HPF_DISTRIBUTION(DISTRIBUTEE, AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE, PLB, PUB, PSTRIDE, LOW_SHADOW, HIGH_SHADOW)

省略可能な引数. AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE, PLB, PUB, PSTRIDE, LOW_SHADOW, HIGH_SHADOW.

機能. 変数の最終 *align-target* の分散に関する情報を返す。

分類. マッピング問合せサブルーチン

引数.

DISTRIBUTEE どの型でもよく、スカラでも配列でもよい。連続であってはならない。空状態のポインタ又は割り付けられていない割付け配列であってはならない。INTENT (IN) 引数とする。

AXIS_TYPE (省略可能) 基本文字型の 1 次元配列でなければならない。長さは任意であるが、完全な値を得るためには少なくとも長さが 9 必要である。各要素には以下の値が文字組込み代入文と同じ方法で設定される。少なくとも

DISTRIBUTEEが最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは HPF_TEMPLATE が TEMPLATE_RANK に返す値である)。INTENT (OUT) 引数とする。i 番目の要素には、*align-target* の i 番目の次元軸の分散に関する情報が設定される。HPF では以下の値が定義されている (実装によってはその他の値が定義されていることもある)。

'BLOCK' この次元軸は BLOCK 分散されている。AXIS_INFO の対応する要素にはブロックの大きさが設定される。

'GEN_BLOCK' この次元軸は、GEN_BLOCK(array) [1]で分散されている。AXIS_INFO の対応する要素の値は実装依存とする。

'COLLAPSED' この次元軸は縮退している (「*」指定により分散されている)。AXIS_INFO の対応する要素の値は実装依存とする。

'CYCLIC' この次元軸は CYCLIC 分散されている。AXIS_INFO の対応する要素にはブロックの大きさが設定される。

'INDIRECT' この次元軸は、INDIRECT(map-array) で分散されている。AXIS_INFO の対応する要素の値は実装依存とする。

AXIS_INFO (省略可能) 基本整数型の 1 次元配列でなければならない。少なくともDISTRIBUTEEが最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは HPF_TEMPLATE が TEMPLATE_RANK に返す値である)。INTENT (OUT) 引数とする。AXIS_INFO の i 番目の要素には、DISTRIBUTEE の最終 *align-target* の i 番目の次元軸の、ブロック分散又はサイクリック分散でのブロックの大きさが設定される。次元軸が縮退しているとき、設定される値は実装に依存する。

PROCESSORS_RANK (省略可能) 基本整数型スカラでなければならない。DISTRIBUTEE が分散されるプロセッサ構成の次元数が設定される。INTENT (OUT) 引数とする。

PROCESSORS_SHAPE (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも PROCESSORS_RANK の返す値 m に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。最初の m 個の要素に、DISTRIBUTEE がマップされるプロセッサ構成の形状が設定される (HPF_DISTRIBUTION は 2 度呼び出す必要があるかもしれない。すなわち最初の呼び出しで PROCESSORS_RANK の値を得、それにより PROCESSORS_SHAPE を割り付けるのである)。

PLB (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも DISTRIBUTEE の最終 *align-target* の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。PLB の i 番目の要素には、DISTRIBUTEE の最終 *align-target* の i 番目の次元軸がマップされているプロセッサの最小の添字が設定される。次元軸が縮退しているとき、設定される値は実装に依存する。設定される値は 1 から e_i の間とする。ここで e_i は、DISTRIBUTEE の最終 *align-target* の選択された次元軸がマップされているプロセッサ構成の次元軸の寸法とする。

[1] 訳注: 原文では「BLOCK(array) となっているが、誤りと思われる

PUB (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも DISTRIBUTEE の最終 *align-target* の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。PUB の i 番目の要素には、DISTRIBUTEE の最終 *align-target* の i 番目の次元軸がマップされているプロセッサの最大の添字が設定される。次元軸が縮退しているとき、設定される値は実装に依存する。設定される値は 1 から e_i の間とする。ここで e_i は、DISTRIBUTEE の最終 *align-target* の選択された次元軸がマップされているプロセッサ構成の次元軸の寸法とする。

PSTRIDE (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも DISTRIBUTEE の次元数に等しい大きさを持たなければならない。INTENT (OUT) 引数とする。PSTRIDE の i 番目の要素には、DISTRIBUTEE の i 番目の次元軸がマップされる ONTO 節中の、プロセッサ間の刻み幅が設定される。次元軸が縮退しているときは、0 が設定される。

LOW_SHADOW (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも DISTRIBUTEE の最終 *align-target* の次元数に等しい大きさを持たなければならない (これは HPF_TEMPLATE が TEMPLATE_RANK に返す値である)。INTENT (OUT) 引数とする。LOW_SHADOW の i 番目の要素には、DISTRIBUTEE の最終 *align-target* の i 番目の次元軸の、ブロック分散又はサイクリック分散での下方のシャドウ幅が設定される。次元軸が縮退しているとき、設定される値は実装に依存する。

HIGH_SHADOW (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも DISTRIBUTEE の最終 *align-target* の次元数に等しい大きさを持たなければならない (これは HPF_TEMPLATE が TEMPLATE_RANK に返す値である)。INTENT (OUT) 引数とする。HIGH_SHADOW の i 番目の要素には、DISTRIBUTEE の最終 *align-target* の i 番目の次元軸の、ブロック分散又はサイクリック分散での上方のシャドウ幅が設定される。次元軸が縮退しているとき、設定される値は実装に依存する。

例示. HPF_ALIGNMENT の例に示す宣言が与えられ、実際のマッピングが宣言通りに行なわれたとすると、HPF_DISTRIBUTE の呼出しの結果は以下のようになる。

	A	B	PI
AXIS_TYPE	['BLOCK', 'BLOCK']	['CYCLIC', 'BLOCK']	[ ]
AXIS_INFO	[10, 10]	[1, 15]	[ ]
PROCESSORS_SHAPE	[4, 2]	[2, 2]	[ ]
PROCESSORS_RANK	2	2	0
PLB	[4, 1]	[2, 1]	[ ]
PUB	[1, 2]	[3, 2]	[ ]
PSTRIDE	[-1,1]	[1, 1]	[ ]

HPF_TEMPLATE(ALIGNEE, TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED, DYNAMIC)

省略可能な引数. LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED, TEMPLATE_RANK, DYNAMIC

機能. 変数の最終 *align-target* に関する情報を返す。HPF_ALIGNMENT が変数の観点からの情報を返すのに対し、HPF_TEMPLATE は最終 *align-target* の観点からの変数に関する情報を返す。

分類. マッピング問合せサブルーチン

引数.

ALIGNEE どの型でもよく、スカラでも配列でもよい。大きさ引継ぎ配列であってはならない。空状態のポインタ又は割り付けられていない割付け配列であってはならない。INTENT (IN) 引数とする。

ALIGNEE がポインタであるときは、指示先の整列に関する情報が返されるものとする。指示先は大きさ引継ぎの仮引数又は大きさ引継ぎの仮引数の部分配列であってはならない。

TEMPLATE_RANK (省略可能) 基本整数型スカラでなければならない。INTENT (OUT) 引数とする。最終 *align-target* の次元数が設定される。この値は、縮退や複製により ALIGNEE の次元数と異なることがある。

LB (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE が最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは TEMPLATE_RANK に設定される値である)。INTENT (OUT) 引数とする。LB の i 番目の要素には、テンプレートの i 番目の次元軸に対する *align-target* の宣言時の下限が設定される。

UB (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE が最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは TEMPLATE_RANK に設定される値である)。INTENT (OUT) 引数とする。UB の i 番目の要素には、テンプレートの i 番目の次元軸に対する *align-target* の宣言時の上限が設定される。

AXIS_TYPE (省略可能) 基本文字型の 1 次元配列でなければならない。長さは任意であるが、完全な値を得るためには少なくとも長さが 10 必要である。各要素には以下の値が文字組込み代入文と同じ方法で設定される。少なくとも ALIGNEE が最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは TEMPLATE_RANK に設定される値である)。INTENT (OUT) 引数とする。AXIS_TYPE の i 番目の要素には、*align-target* の i 番目の次元軸に関する情報が設定される。HPF では以下の値が定義されている (実装によりその他の値が定義されていることがある)。

'NORMAL' *align-target* のこの次元軸には、ALIGNEE のある次元軸が整列している。この値が代入された AXIS_TYPE の要素に対しては、対応する AXIS_INFO の要素に、この *align-target* の次元軸に整列する ALIGNEE の次元軸の番号が設定される。

'REPLICATED' *align-target* のこの次元軸には、ALIGNEE が複製されている。この値が代入された AXIS_TYPE の要素に対しては、対応する AXIS_INFO の要素に、この *align-target* の次元軸に複製された ALIGNEE の複製の数が設定される。

'SINGLE' *align-target* のこの次元軸のある座標に、ALIGNEE が整列している。この値が代入された AXIS_TYPE の要素に対しては、対応する AXIS_INFO の要素に、ALIGNEE が整列している *align-target* の座標が設定される。

AXIS_INFO (省略可能) 基本整数型の 1 次元配列でなければならない。少なくとも ALIGNEE が最終的に整列している *align-target* の次元数と等しい大きさを持たなければならない (これは TEMPLATE_RANK に設定される値である)。INTENT (OUT) 引数とする。上記 AXIS_TYPE の項を参照されたい。

NUMBER_ALIGNED (省略可能) 基本整数型スカラでなければならない。INTENT (OUT) 引数とする。この最終 *align-target* に整列している変数の総数が設定される。これは、この *align-target* が再分散されたときに同時に移動する変数の数である。

DYNAMIC (省略可能) 基本論理型スカラでなければならない。INTENT (OUT) 引数とする。*align-target* が DYNAMIC 属性を持つとき、真が設定される。それ以外のとき、偽が設定される。

例示. HPF_ALIGNMENT の例に示す宣言が与えられ、実際のマッピングが宣言通りに行なわれたとすると、HPF_TEMPLATE の呼出しの結果は以下のようになる。

	A	C	D
LB	[1, 1]	[1, 1]	[1, 1]
UB	[40, 20]	[40, 20]	[40, 20]
AXIS_TYPE	['NORMAL', 'NORMAL']	['NORMAL', 'NORMAL']	['NORMAL', 'SINGLE']
AXIS_INFO	[1, 2]	[3, 1]	[1, 4]
NUMBER_ALIGNED	3	3	3
TEMPLATE_RANK	2	2	2
DYNAMIC	false	false	false

HPF_MAP_ARRAY(ARRAY, TEMPLATE_DIM, MAP_ARRAY)

機能. ARRAY の最終 *align-target* の第 TEMPLATE_DIM 次元について、間接分散で用いられるマップ配列を返す。

分類. マッピング問合せサブルーチン

引数.

ARRAY どの型でもよい。スカラであってはならない。連続であってはならない。空状態のポインタ又は割り付けられていない割付け配列であってはならない。INTENT (IN) 引数とする。

TEMPLATE_DIM 基本整数型スカラでなければならない。1 と ARRAY の最終 *align-target* の次元数の間の値でなければならない。INTENT (IN) 引数とする。

MAP_ARRAY 基本文字型の 1 次元配列でなければならない。ARRAY の最終 *align-target* が分散しているプロセッサ構成の、第 PROCESSORS_DIM 次元の寸法以上の大きさを持たなければならない。INTENT (OUT) 引数とする。

MAP_ARRAY の i 番目の要素には、ARRAY の最終 *align-target* の第 TEMPLATE_DIM 次元の、i 番目の要素がマップされるプロセッサの添字が設定される。ARRAY の最終 *align-target* の第 TEMPLATE_DIM 次元が縮退しているとき、すべての要素に 1 が設定される。

例示. 以下の宣言が与えられ、

```
      DIMENSION A(2)
!HPF$ TEMPLATE T(4,8)
!HPF$ ALIGN A(I,*) WITH T(2*I,5)
!HPF$ PROCESSORS PROCS(2,2)
!HPF$ DISTRIBUTE T(INDIRECT( (/1,2,2,1/) ), BLOCK( (/3,5/) )) ONTO PROCS
```

実際のマッピングが宣言通りに行なわれたとすると、HPF_MAP_ARRAY(A,TEMPLATE_DIM=1, MAP_ARRAY=M) の呼出しの結果、M の値は $\begin{bmatrix} 1 & 2 & 2 & 1 \end{bmatrix}$
になる。また HPF_MAP_ARRAY(A,TEMPLATE_DIM=2, MAP_ARRAY=M) の呼出しの結果、M の値は $\begin{bmatrix} 1 & 1 & 1 & 2 & 2 & 2 & 2 & 2 \end{bmatrix}$ になる。

HPF_NUMBER_MAPPED(ARRAY, PROCESSORS_DIM, NUMBER_MAPPED)

機能. ARRAY の最終 *align-target* が分散されるプロセッサ構成の、第 PROCESSORS_DIM 次元の各要素にマップされる、ARRAY の最終 *align-target* の要素数を返す。

分類. マッピング問合せサブルーチン

引数.

ARRAY どの型でもよい。スカラであってはならない。連続であってはならない。空状態のポインタ又は割り付けられていない割付け配列であってはならない。

PROCESSORS_DIM 基本整数型スカラでなければならない。1 と ARRAY の最終 *align-target* が分散されるプロセッサ構成の次元数との間の値でなければならない。INTENT (IN) 引数とする。

NUMBER_MAPPED 基本整数型の 1 次元配列でなければならない。ARRAY の最終 *align-target* が分散しているプロセッサ構成の、第 PROCESSORS_DIM 次元の寸法以上の大きさを持たなければならない。NUMBER_MAPPED の i 番目の要素には、ARRAY の最終 *align-target* が分散されるプロセッサ構成の、第 PROCESSORS_DIM 次元の i 番目のプロセッサにマップされる、ARRAY の最終 *align-target* の対応する次元の要素数が設定される。ARRAY の最終 *align-target* が分散されるプロセッサ構成の第 PROCESSORS_DIM 次元が、BLOCK 分散の次元と対応づけられているとき、MAP_ARRAY にはその次元を分散するのに用いられるブロックの大きさの配列が設定される。

例示. 以下の宣言が与えられ、

```
      DIMENSION A(2,40)
!HPF$ TEMPLATE T(4,8,4,16)
!HPF$ ALIGN A(I,*) WITH T(2*I, 5, *, *)
!HPF$ PROCESSORS PROCS(2,2,3)
!HPF$ DISTRIBUTE T(INDIRECT((/2,2,1,2/)),  BLOCK((/3,5/)), *, BLOCK) &
!HPF$      ONTO PROCS
```

実際のマッピングが宣言通りに行なわれたとすると、HPF_NUMBER_MAPPED(A,PROCESSORS_DIM=1, NUMBER_MAPPED = M) の呼出しの結果、M の値は $\begin{bmatrix} 1 & 3 \end{bmatrix}$ になり、HPF_NUMBER_MAPPED(A,PROCESSORS_DIM=2, NUMBER_MAPPED = M) の呼出しの結果、M の値は $\begin{bmatrix} 3 & 5 \end{bmatrix}$ になり、HPF_NUMBER_MAPPED(A,PROCESSORS_DIM=3, NUMBER_MAPPED = M) の呼出しの結果、M の値は $\begin{bmatrix} 6 & 6 & 4 \end{bmatrix}$ になる。

第IV部

附属書

ここでは、High Performance Fortran 言語 バージョン 2.0 (第 I 部と第 II 部で記述) および、公認拡張仕様 (第 III 部で記述) の機能の文法と意味について記述を、参照利用のためにまとめる。ここは、厳密な意味での HPF 言語仕様書の一部ではない。

附属書 A　構文規則

この附属書は、本 High Performance Fortran 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

H201　*hpf-directive-line*　**is**　*directive-origin*　*hpf-directive*

H202　*directive-origin*　**is**　`!HPF$`
or　`CHPF$`
or　`*HPF$`

H203　*hpf-directive*　**is**　*specification-directive*
or　*executable-directive*

H204　*specification-directive*　**is**　*processors-directive*
or　*align-directive*
or　*distribute-directive*
or　*inherit-directive*
or　*template-directive*
or　*combined-directive*
or　*sequence-directive*

H205　*executable-directive*　**is**　*independent-directive*

制約:　*hpf-directive-line* には、他の文があってはならない。

制約:　*specification-directive* は、宣言構文 (*declaration-construct*) が現れてもよい場所にのみ現れてもよい。

制約:　*executable-directive* は、実行構文 (*executable-construct*) が現れてもよい場所にのみ現れてもよい。

制約:　*hpf-directive-line* は、プログラム単位内の周囲の形式に従って、Fortran の自由形式 (F95:3.3.1.1) または固定形式 (F95:3.3.2.1) の注釈行の規則のどちらかに従う。(F95:3.3)

H206 *specification-directive-extended* **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 *range-directive*
or *shadow-directive*

H207 *executable-directive-extended* **is** *independent-directive*
or *realign-directive*
or *redistribute-directive*
or *on-directive*
or *resident-directive*

H208 *executable-construct-extended* **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*

A.3 データマッピング

A.3.2 データ整列とデータ分散指示文の構文

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*

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。

A.3.3 DISTRIBUTE 指示文

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 *

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。

A.3.4 ALIGN 指示文

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。

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*

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。

A.3.6 PROCESSORS 指示文

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

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

A.3.7 TEMPLATE 指示文

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

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

A.3.8 記憶列結合と順序結合

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。

A.4 副プログラム境界でのデータマッピング

A.4.4 整列

H401 *inherit-directive* **is** `INHERIT` *inheritee-list*

H402 *inheritee* **is** *object-name*

制約: *inheritee* は仮引数でなければならない。

制約: *inheritee* は *alignee* であってはならない。

制約: *inheritee* は *distributee* であってはならない。

A.5 INDEPENDENT 指示文及び関連の指示文

A.5.1 INDEPENDENT 指示文

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。

[1] 原文は *reduction-var*

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。

A.6 外来プログラム単位

A.6.2 外来プログラム単位の宣言

H601 *function-stmt*
is [*prefix*] FUNCTION *function-name*
([*dummy-arg-name-list*])
[RESULT (*result-name*)]

H602 *subroutine-stmt*
is [*prefix*] SUBROUTINE *subroutine-name*
[([*dummy-arg-list*])]

H603 *prefix*
is *prefix-spec* [*prefix-spec*] ...

H604 *prefix-spec*
is *type-spec*
or RECURSIVE
or PURE
or ELEMENTAL
or *extrinsic-prefix*

制約: 任意の HPF の外部副プログラム内で、どの内部副プログラムも、その親と同じ外来種別でなければならず、また、明示的に外来種別が与えられていない内部副プログラムに対しては、その親と同じ外来種別が仮定される。

H605 *program-stmt*
is [*extrinsic-prefix*] PROGRAM *program-name*

H606 *module-stmt* **is** [*extrinsic-prefix*] MODULE *module-name*

H607 *block-data-stmt* **is** [*extrinsic-prefix*] BLOCK DATA
[*block-data-name*]

制約: 任意の HPF のモジュール内で、どのモジュール副プログラムも、その親と同じ外来種別でなければならず、また明示的に外来種別が与えられていないモジュール副プログラムに対しては、その親と同じ外来種別が仮定される。

制約: HPF の任意の主プログラムあるいはモジュール副プログラム内で、どの内部副プログラムも、その親と同じ外来種別でなければならず、また明示的に外来種別が与えられていない内部副プログラムに対しては、その親と同じ外来種別が仮定される。

H608 *extrinsic-prefix* **is** EXTRINSIC (*extrinsic-spec*)

H609 *extrinsic-spec* **is** *extrinsic-spec-arg-list*
or *extrinsic-kind-keyword*

H610 *extrinsic-spec-arg* **is** *language*
or *model*
or *external-name*

H611 *language* **is** [LANGUAGE =]
scalar-char-initialization-expr

H612 *model* **is** [MODEL =]
scalar-char-initialization-expr

H613 *external-name* **is** [EXTERNAL_NAME =]
scalar-char-initialization-expr

制約: *extrinsic-spec-arg-list* の中には、*language*、*model*、あるいは *external-name* の少なくとも 1 つが指定されなければならず、またどれも 2 回以上指定することはできない。

制約: もし、LANGUAGE=を使わずに *language* を指定する場合、*language* は *extrinsic-spec-arg-list* 中の最初の要素でなければならない。もし、MODEL=を使わずに *model* を指定する場合、LANGUAGE=のない *language* が *extrinsic-spec-arg-list* 中の最初の要素であり、*model* が 2 番目の要素でなければならない。もし、EXTERNAL_NAME=を使わずに *external-name* を指定する場合、LANGUAGE=のない *language* が *extrinsic-spec-arg-list* 中の最初の要素であり、MODEL=のない *model* が 2 番目の要素でなければならない。

制約: LANGUAGE=、MODEL=、EXTERNAL_NAME=を伴う形は、上で禁止された場合を除いて、いかなる順番で書いても構わない。

これらの *extrinsic-spec-arg-list* に関する規則は、あたかも EXTRINSIC が、LANGUAGE、MODEL、EXTERNAL_NAME という、それぞれが OPTIONAL であるような *dummy-arg-list* を使用した明示的引用仕様を伴った手続であるかのようなものであることに注意されたい。

制約: *language* の中では、*char-initialization-expr* の値は、以下のものが許される。

- 'HPF'　HPF 言語を指す。もし *model* が明示的に指定されていない場合、*model* には'GLOBAL'が暗黙に仮定される。
- 'FORTRAN' ANSI または ISO 規格の Fortran 言語を指す。もし *model* が明示的に指定されていない場合、*model* には'SERIAL'が暗黙に仮定される。
- 'F77' 以前の ANSI または ISO 規格である FORTRAN 77 言語を指す。もし *model* が明示的に指定されていない場合、*model* には'SERIAL'が暗黙に仮定される。
- 'C' ANSI 規格の C 言語を指す。もし *model* が明示的に指定されていない場合、*model* には'SERIAL'が暗黙に仮定される。
- 実装依存の値。暗黙の *model* は実装に依存する。

ほとんどの実装にとって、'C'は、引用仕様本体 (*interface-body*) 中に記述された FUNCTION 文か SUBROUTINE 文でしか許されないことに注意されたい。

制約: *language* が指定されていない場合、親有効域と同じものが仮定される。

制約: *model* において、*char-initialization-expr* の値は、以下のものが許される。

- 'GLOBAL' グローバルモデルを指す。
- 'LOCAL' ローカルモデルを指す。
- 'SERIAL' シリアルモデルを指す。
- 実装依存の値。

制約: *model* が指定されず、*language* の指定から暗黙に仮定されない場合、親有効域と同じものが仮定される。

制約: 名称が HPF の 3 文字から始まる全ての *language* 及び *model* は、本仕様及びその後継仕様の現在あるいは将来における定義のために予約されている。

制約: *external-name* において、*scalar-char-initialization-expr* の値は、その用途が外来種別によって決定される文字列である。例えば、ある外来種別は、*external-name* を、その手続が C の手続から参照された場合の名前を指定するために使用するかもしれない。そのような実装では、ユーザはコンパイラに、その名前が C コンパイラに理解できるように変換を行なうことを期待するであろう。もし *external-name* が指定されていない場合、その値は実装依存となる。

H614 *extrinsic-kind-keyword* **is** HPF
or HPF_LOCAL
or HPF_SERIAL

制約: EXTRINSIC(HPF) は EXTRINSIC('HPF','GLOBAL') と同値である。*extrinsic-prefix* が存在しないとき、HPF コンパイラはコンパイル単位を外来種別 HPF に属するかのように解釈する。従って、HPF コンパイラにとって、EXTRINSIC(HPF) あるいは

EXTRINSIC('HPF', GLOBAL') と指定するのは冗長である。しかし、このような明示的な指定は、複数の外来種別をサポートしているコンパイラを使用する場合に必要となるかもしれない。

制約: EXTRINSIC(HPF_LOCAL) は EXTRINSIC('HPF','LOCAL') と同値である。外来種別が HPF_LOCAL であるような主プログラムは、外来種別が HPF で、実行部がそのサブルーチンの呼出しだけから構成された主プログラムから引数なしで呼び出される、外来種別 HPF_LOCAL のサブルーチンであるかのように振る舞う。

制約: EXTRINSIC(HPF_SERIAL) は EXTRINSIC('HPF','SERIAL') と同値である。外来種別が HPF_SERIAL であるような主プログラムは、外来種別が HPF で、実行部分がそのサブルーチンの呼出しだけから構成された主プログラムから引数なしで呼び出される、外来種別 HPF_SERIAL のサブルーチンであるかのように振る舞う。

制約: 名称が HPF の 3 文字から始まる全ての *extrinsic-kind-keyword* は、本仕様及びその後継仕様の現在あるいは将来における定義のために予約されている。

A.8 データマッピングの公認拡張仕様

A.8.2 拡張データマッピング指示文の属性形式の構文

H801 *combined-attribute-extended* **is** ALIGN *align-attribute-stuff*
or DISTRIBUTE *dist-attribute-stuff*
or INHERIT
or TEMPLATE
or PROCESSORS
or DIMENSION (*explicit-shape-spec-list*)
or DYNAMIC
or RANGE *range-attr-stuff*
or SHADOW *shadow-attr-stuff*
or SUBSET

制約: SUBSET 属性は、プロセッサ構成にだけ適用できる。

A.8.3 REDISTRIBUTE 指示文

H802 *redistribute-directive* **is** REDISTRIBUTE *distributee dist-directive-stuff*
or REDISTRIBUTE *dist-attribute-stuff* ::
distributee-list

制約: REDISTRIBUTE 指示文に現れる *distributee* は、DYNAMIC 属性をもたなければならない (8.5 節 参照)。

制約: REDISTRIBUTE 指示文の *distributee* は、ALIGN または REALIGN 指示文の *alignee* として現れてはならない。

制約: REDISTRIBUTE 指示文の *dist-format-clause* や *dist-target* のどちらも、“*” で始まってはならない。

A.8.4 REALIGN 指示文

H803 *realign-directive* **is** REALIGN *alignee align-directive-stuff*
or REALIGN *align-attribute-stuff* :: *alignee-list*

制約: REALIGN 指示文に現れる *alignee* は、DYNAMIC 属性をもたなければならない (8.5 節参照)。

制約: *align-with-clause* に指定された *align-target* が DYNAMIC 属性をもつ場合、*alignee* も同様に DYNAMIC 属性をもたなければならない。

制約: REALIGN 指示文の *alignee* は、DISTRIBUTE や REDISTRIBUTE 指示文の *distributee* であってはならない。

A.8.5 DYNAMIC 指示文

H804 *dynamic-directive* **is** DYNAMIC *alignee-or-distributee-list*

H805 *alignee-or-distributee* **is** *alignee*
or *distributee*

制約: 共通ブロックの実体は、DYNAMIC と宣言することはできず、また、DYNAMIC である実体 (またはテンプレート) に整列することはできない。(ここで制約されているようなことをしたいならば、共通ブロックの代わりにモジュールを使わなければならない。)

制約: 構造型の成分は、POINTER 属性をもつ場合だけ DYNAMIC 属性をもつことができる。(詳しくは、8.9 節を参照されたい。)

制約: SAVE 属性をもつ実体は、DYNAMIC と宣言することはできず、また、DYNAMIC である実体 (またはテンプレート) に整列することはできない。

A.8.7 部分プロセッサへのマッピング

H806 *extended-dist-target* **is** *processors-name* [(*section-subscript-list*)]
or * *processors-name* [(*section-subscript-list*)]
or *

制約: 部分配列添字並び (*section-subscript-list*) の部分配列添字 (*section-subscript*) は、ベクトル添字 (*vector-subscript*) であってはならず、添字 (*subscript*) または添字三つ組 (*subscript-triplet*) でなければならない。

制約: 部分配列添字並び (*section-subscript-list*) では、部分配列添字 (*section-subscript*) の数は、*processor-name* の次元数と等しくなければならない。

制約: DISTRIBUTE 指示文の中では、それぞれの部分配列添字 (*section-subscript*) は、宣言式 (*specification-expr*) でなければならない。

制約: DISTRIBUTE および REDISTRIBUTE 指示文の中で、*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、指定されたプロセッサ構成の添字三つ組 (*subscript-triplet*) の数と等しくなければならない。

制約: DISTRIBUTE および REDISTRIBUTE 指示文の中で、*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、指定されたプロセッサ構成の添字三つ組 (*subscript-triplet*) の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、*distributee* が POINTER 属性をもつ場合を除き、仮引数でなければならない。

制約: ALIGN 指示文の *align-spec* が “*” で始まるときは、すべての *alignee* は、*alignee* が POINTER 属性をもつ場合を除き、仮引数でなければならない。

制約: *inheritee* は、*alignee* が POINTER 属性をもつ場合を除き、仮引数でなければならない。

A.8.9 構造体成分のマッピング

H807 *distributee-extended* **is** *object-name*
or *template-name*
or *component-name*
or *structure-component*

制約: 構造型の成分は、その型が明示的にマップされていない場合に限って明示的に分散することができる。

制約: 構造型のデータ実体は、その構造型が明示的にマップされた型でない場合に限って明示的に分散することができる。

制約: DISTRIBUTE 指示文の *distributee* は、構造体成分 (*structure-component*) であってはならない。

制約: 構造型定義の中に現れる DISTRIBUTE 指示文の *distributee* は、構造型の成分の成分名 (*component-name*) でなければならない。

制約: 構造型定義の中に現れる DISTRIBUTE 指示文に限って、*distributee* を成分名 (*component-name*) とすることができる。

制約: REDISTRIBUTE 指示文の中でだけ *distributee* を構造体成分にすることができる。このとき、右端を除くすべての部分参照はスカラ (0 次元) でなければならない。構造体成分の右端の部分参照は、DYNAMIC 属性をもっていなければならない。

H808 *alignee-extended* **is** *object-name*
or *component-name*
or *structure-component*

制約: 構造型の成分は、その型が明示的にマップされていない場合に限って明示的に整列することができる。

制約: 構造型のデータ実体は、その構造型が明示的にマップされた型でない場合に限って明示的に整列することができる。

制約: ALIGN 指示文の *alignee* は、構造体成分 (*structure-component*) であってはならない。

制約: 構造型定義の中に現れる ALIGN 指示文の *alignee* は、構造型の成分の成分名 (*component-name*) でなければならない。

制約: 構造型定義の中に現れる ALIGN 指示文に限って、*alignee* を成分名 (*component-name*) とすることができる。

制約: REALIGN 指示文の中でだけ *alignee* を構造体成分 (*structure-component*) にすることができる。このとき、右端を除くすべての部分参照はスカラ (0 次元) でなければならない。構造体成分の右端の部分参照は、DYNAMIC 属性をもっていなければならない。

H809 *align-target-extended* **is** *object-name*
or *template-name*
or *component-name*
or *structure-component*

制約: 構造型定義の中に現れる ALIGN 指示文に限って、整列先を成分名 (*component-name*) とすることができる。

制約: *align-target* が構造体成分 (*structure-component*) であるとき、その右端を除くすべての部分参照はスカラ (0 次元) でなければならない。

A.8.10 新しい分散形式

H810 *extended-dist-format* **is** BLOCK [(*int-expr*)]
or CYCLIC [(*int-expr*)]
or GEN_BLOCK (*int-array*)
or INDIRECT (*int-array*)
or *

制約: DISTRIBUTE 指示文または REDISTRIBUTE 指示文の *extended-dist-format* の中に現れる整数型配列 (*int-array*) は、整数型の一次元配列でなければならない。

制約: DISTRIBUTE 指示文の *extended-dist-format* の中に現れる整数型配列 (*int-array*) は、制限式 (*restricted-expr*) でなければならない。

制約: GEN_BLOCK 分散に現れる整数型配列 (*int-array*) の大きさは、分散先のプロセッサ構成の対応する次元の寸法と等しくなければならない。

制約: INDIRECT 分散に現れる整数型配列 (*int-array*) の大きさは、その分散が適用される *distributee* の対応する次元の寸法と等しくなければならない。

A.8.11 RANGE 指示文

H811 *range-directive* **is** RANGE *ranger range-attr-stuff*

H812 *ranger* **is** *object-name*
or *template-name*

H813 *range-attr-stuff* **is** *range-distribution-list*

H814 *range-distribution* **is** (*range-attr-list*)

H815 *range-attr* **is** *range-dist-format*
or ALL

H816 *range-dist-format* **is** BLOCK [( )]
or CYCLIC [( )]
or GEN_BLOCK
or INDIRECT
or *

制約: 少なくとも以下のいずれかが成り立たなければならない。

- *ranger* は DYNAMIC 属性をもつ。
- *ranger* は INHERIT 属性をもつ。
- *ranger* は DISTRIBUTE 指示文か *combined-directive* で指定され、その *dist-format-clause* は*である。

制約: *range-attr-list* の長さはそれぞれ、*ranger* の次元数と等しくなければならない。

制約: *ranger* は ALIGN 指示文または REALIGN 指示文の *alignee* であってはならない。

A.8.12 SHADOW 指示文

H817 *shadow-directive* **is** SHADOW *shadow-target shadow-attr-stuff*

H818 *shadow-target* **is** *object-name*
or *component-name*

H819 *shadow-attr-stuff* **is** (*shadow-spec-list*)

H820 *shadow-spec* **is** *width*
or *low-width* : *high-width*

H821 *width* **is** *int-expr*

H822 *low-width* **is** *int-expr*

H823 *high-width* **is** *int-expr*

制約: *width*、 *low-width* または *high-width* として現れる整数式 (*int-expr*) は、0 以上の値をもつ宣言式 (*specification-expr*) でなければならない。

A.9 データとタスク並列に対する公認拡張

A.9.1 活動プロセッサ集合

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

A.9.2 ON 指示文

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

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

H904 *on-construct* **is**

directive-origin block-on-directive
block
directive-origin end-on-directive

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

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

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

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

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

A.9.3 RESIDENT 節、指示文、構文

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

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

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

H913 *resident-construct* **is**

directive-origin block-resident-directive
block
directive-origin end-resident-directive

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

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

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

A.9.4 TASK_REGION 構文

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

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

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

A.10 非同期入出力に関する公認拡張

or `ASYNCHRONOUS`

or `ID =` *scalar-default-int-variable*
or `ASYNCHRONOUS`

制約: `ASYNCHRONOUS` 指定子又は `ID=`指定子のどちらか一方を指定した場合、その両方を指定しなければならない。

制約: `ASYNCHROUNOUS` 指定子を指定した場合、`REC=`指定子を指定しなければならず、かつ「書式」を指定してはならず、かつ「変数群名」を指定してはならない。

制約: `ASYNCHRONOUS` 指定子を指定した場合、データ転送文中のどの式においても、関数を引用してはならない。

or `ID =` *scalar-default-int-variable*
or `PENDING =` *scalar-default-logical-variable*

制約: `INQUIRE` 文中に `FILE=`指定子を指定した場合、`ID=`指定子や `PENDING=`指定子を指定してはならない。

制約: `ID=`指定子又は `PENDING=`指定子のどちらか一方を指定する場合、その両方を指定しなければならない。

A.10.1 WAIT 文

H1001 *wait-stmt* **is** `WAIT (` *wait-spec-list* `)`

H1002 *wait-spec* **is** `UNIT =` *io-unit*
or `ID =` *scalar-default-int-expr*
or `ERR =` *label*
or `IOSTAT =` *label*

制約: *wait-spec-list* には、UNIT=指定子をちょうど 1 つ、ID=指定子をちょうど 1 つと、他の各指定子を高々1 つ、指定しなければならない。

A.11 HPF 外来機能に関する公認拡張

A.11.2 外来言語との結合仕様

H1101 *type-declaration-stmt-extended* **is** *type-spec* [[, *attr-spec-extended*] ... ::] *entity-decl-list*

H1102 *attr-spec-extended*
is PARAMETER
or *access-spec*
or ALLOCATABLE
or DIMENSION (*array-spec*)
or EXTERNAL
or INTENT (*intent-spec*)
or INTRINSIC
or OPTIONAL
or POINTER
or SAVE
or TARGET
or MAP_TO (*map-to-spec*)
or LAYOUT (*layout-spec*)
or PASS_BY (*pass-by-spec*)

H1103 *map-to-spec* **is** *scalar-char-initialization-expr*

H1104 *layout-spec* **is** *scalar-char-initialization-expr*

H1105 *pass-by-spec* **is** *scalar-char-initialization-expr*

制約: 同じ *attr-spec-extended* を、一つの *type-declaration-stmt* 中で 2 回以上指定してはならない。

制約: 一つのデータ要素には、一つの有効域内で、どの属性も 2 回以上明示的に指定してはならない。

制約: MAP_TO 属性、LAYOUT 属性、及び PASS_BY 属性は、これらの属性が明示的に定義されている外来種別の有効域内で、仮引数に対してだけ指定できる。

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。H で始まる識別番号は、本 High Performance Fortran 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。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	H208
add-op	R710	H323 H505
add-operand	R706	H326 H505
align-add-operand	H324	H323 H324
align-attribute-stuff	H315	H302 H801 H803
align-directive	H313	H204 H206
align-directive-stuff	H314	H313 H803
align-dummy	H318	H317 H325
align-primary	H325	H324
align-source	H317	H314 H315
align-spec	H320	H319
align-subscript	H322	H320
align-subscript-use	H323	H322 H323 H325
align-target	H321	H320
align-target-extended	H809	
align-with-clause	H319	H314 H315
alignee	H316	H313 H803 H805
alignee-extended	H808	
alignee-or-distributee	H805	H804
allocate-object	R625	
allocate-stmt	R622	
and-op	R720	H505
and-operand	R715	H505
array-constructor	R432	
array-spec	R513	H1102
assignment-stmt	R735	
association-name	H334	H333
attr-spec	R503	

attr-spec-extended	H1102	H1101				
block	R801	H904	H913	H917		
block-data-stmt	H607					
block-on-directive	H905	H904				
block-resident-directive	H914	H913				
block-task-region-directive	H918	H917				
call-stmt	R1211					
case-construct	R808	H208				
combined-attribute	H302	H301				
combined-attribute-extended	H801					
combined-decl	H303	H301				
combined-directive	H301	H204	H206			
data-stmt	R532					
deallocate-stmt	R631					
directive-origin	H202	H201	H904	H913	H917	
dist-attribute-stuff	H307	H302	H801	H802		
dist-directive-stuff	H306	H305	H307	H802		
dist-format	H310	H309				
dist-format-clause	H309	H306				
dist-onto-clause	H311	H306	H307			
dist-target	H312	H311				
distribute-directive	H305	H204	H206			
distributee	H308	H305	H802	H805		
distributee-extended	H807					
do-construct	R816	H208				
dummy-arg	R1223	H602				
dynamic-directive	H804	H206				
end-function-stmt	R1220					
end-on-directive	H906	H904				
end-resident-directive	H915	H913				
end-subroutine-stmt	R1224					
end-task-region-directive	H919	H917				
entity-decl	R504	H1101				
equiv-op	R722	H505				
equiv-operand	R717	H505				
executable-construct	R215					
executable-construct-extended	H208					
executable-directive	H205	H203				
executable-directive-extended	H207					
execution-part	R208					
explicit-shape-spec	R514	H302	H303	H330	H332	H801
expr	R723	H505				

extended-dist-format	H810	
extended-dist-target	H806	
external-name	H613	H610
extrinsic-kind-keyword	H614	H609
extrinsic-prefix	H608	H604 H605 H606 H607
extrinsic-spec	H609	H608
extrinsic-spec-arg	H610	H609
function-reference	R1210	
function-stmt	H601	
function-subprogram	R1216	
high-width	H823	H820
home	H907	H903
hpf-directive	H203	H201
hpf-directive-line	H201	
hpf-entity	H304	H303
if-construct	R802	H208
independent-directive	H501	H205 H207
inherit-directive	H401	H204 H206
inheritee	H402	H401
input-item	R914	
int-add-operand	H326	H323 H324
int-expr	R728	H310 H322 H810 H821 H822 H823
int-level-two-expr	H328	H323
int-mult-operand	H327	H324
int-variable	R607	H318
interface-body	R1205	
internal-subprogram-part	R210	
io-unit	R901	H1002
kind-selector	R506	
label	R313	H1002
language	H611	H610
layout-spec	H1104	H1102
level-2-expr	R707	H328 H505
low-width	H822	H820
map-to-spec	H1103	H1102
mask-expr	R743	
model	H612	H610
module-stmt	H606	
mult-op	R709	H505
mult-operand	R705	H327 H505
namelist-stmt	R544	
new-clause	H502	H501 H903

nullify-stmt	R629				
on-construct	H904	H208			
on-directive	H902	H207			
on-stuff	H903	H902	H905		
or-op	R721	H505			
or-operand	R716	H505			
output-item	R915				
pass-by-spec	H1105	H1102			
pointer-assignment-stmt	R736				
pointer-object	R630				
prefix	H603	H601	H602		
prefix-spec	H604	H603			
processors-decl	H330	H329			
processors-directive	H329	H204	H206		
processors-elmt	H909	H907			
program-stmt	H605				
range-attr	H815	H814			
range-attr-stuff	H813	H801	H811		
range-directive	H811	H206			
range-dist-format	H816	H815			
range-distribution	H814	H813			
ranger	H812	H811			
read-stmt	R909				
realign-directive	H803	H207			
redistribute-directive	H802	H207			
reduction-clause	H503	H501			
reduction-function	H506	H505			
reduction-stmt	H505				
reduction-variable	H504	H503			
res-object	H916	H911			
resident-clause	H910	H903			
resident-construct	H913	H208			
resident-directive	H912	H207			
resident-stuff	H911	H910	H912	H914	
section-subscript	R618	H806	H908	H909	
sequence-directive	H333	H204	H206		
shadow-attr-stuff	H819	H801	H817		
shadow-directive	H817	H206			
shadow-spec	H820	H819			
shadow-target	H818	H817			
specification-directive	H204	H203			
specification-directive-extended	H206				

specification-expr	R734	
specification-part	R204	
stat-variable	R623	
stop-stmt	R840	
stride	R620	
structure-component	R614	H504 H807 H808 H809
subroutine-stmt	H602	
subscript	R617	
subscript-triplet	R619	H322
subset-directive	H901	H206
target	R737	
task-region-construct	H917	H208
template-decl	H332	H331
template-directive	H331	H204 H206
template-elmt	H908	H907
type-declaration-stmt	R501	
type-declaration-stmt-extended	H1101	
type-spec	R502	H604 H1101
variable	R601	H505 H907
wait-spec	H1002	H1001
wait-stmt	H1001	
where-construct	R739	H208
where-stmt	R738	
width	H821	H820
write-stmt	R910	

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
access-spec	H1102
array-variable-name	H504
block-data-name	H607
char-initialization-expr	H611 H612 H613 H1103 H1104 H1105
common-block-name	H334
component-name	H807 H808 H809 H818
default-int-expr	H1002
dummy-arg-name	H601
function-name	H601
int-array	H810
intent-spec	H1102
module-name	H606

object	H916
object-name	H303 H308 H316 H321 H334 H402 H807 H808 H809 H812 H818
processors-name	H304 H312 H330 H806 H901 H909
program-name	H605
result-name	H601
subroutine-name	H602
template-name	H304 H308 H321 H332 H807 H809 H812 H908
variable-name	H502 H504

B.3 終端記号

記号	参照箇所
!HPF$	H202
(	H302 H303 H309 H310 H314 H315 H320 H325 H330 H332 H502 H503 H505 H601 H602 H608 H801 H806 H810 H814 H816 H819 H907 H908 H909 H911 H1001 H1102
)	H302 H303 H309 H310 H314 H315 H320 H325 H330 H332 H502 H503 H505 H601 H602 H608 H801 H806 H810 H814 H816 H819 H907 H908 H909 H911 H1001 H1102
*	H309 H310 H312 H317 H320 H322 H324 H505 H806 H810 H816
*HPF$	H202
+	H505
,	H501 H505 H903 H1101
/	H334
:	H317 H820
::	H301 H333 H802 H803 H1101
=	H505 H611 H612 H613 H1002
ALIGN	H302 H313 H801
ALL	H815
ALLOCATABLE	H1102
BEGIN	H905 H914
BLOCK	H310 H607 H810 H816
CHPF$	H202
CYCLIC	H310 H810 H816

DATA	H607
DIMENSION	H302 H801 H1102
DISTRIBUTE	H302 H305 H801
DYNAMIC	H801 H804
ELEMENTAL	H604
END	H906 H915 H919
ERR	H1002
EXTERNAL	H1102
EXTERNAL_NAME	H613
EXTRINSIC	H608
FUNCTION	H601
GEN_BLOCK	H810 H816
HOME	H907
HPF	H614
HPF_LOCAL	H614
HPF_SERIAL	H614
IAND	H506
ID	H1002
IEOR	H506
INDEPENDENT	H501
INDIRECT	H810 H816
INHERIT	H302 H401 H801
INTENT	H1102
INTRINSIC	H1102
IOR	H506
IOSTAT	H1002
LANGUAGE	H611
LAYOUT	H1102
MAP_TO	H1102
MAX	H506
MIN	H506
MODEL	H612
MODULE	H606
NEW	H502
NO	H333
ON	H902 H905 H906
ONTO	H311
OPTIONAL	H1102
PARAMETER	H1102
PASS_BY	H1102
POINTER	H1102
PROCESSORS	H302 H329 H801

PROGRAM	H605
PURE	H604
RANGE	H801 H811
REALIGN	H803
RECURSIVE	H604
REDISTRIBUTE	H802
REDUCTION	H503
RESIDENT	H910 H912 H914 H915
RESULT	H601
SAVE	H1102
SEQUENCE	H333
SHADOW	H801 H817
SUBROUTINE	H602
SUBSET	H801 H901
TARGET	H1102
TASK_REGION	H918 H919
TEMPLATE	H302 H331 H801
UNIT	H1002
WAIT	H1001
WITH	H319

附属書C　HPF 1.1 Subset

As part of the definition of the previous version of the High Performance Fortran language, HPF 1.1, a subset language was formally defined, based on the Fortran 77 language. The goal was to permit more rapid implementations of a useful subset of HPF that did not require full implementation of the new ANSI/ISO standard Fortran ("Fortran 90").

No subset language is defined as part of the current version, HPF 2.0. This Annex is included in the HPF 2.0 language document as a convenient summary of the HPF 1.1 Subset, which has served as a minimum requirement for HPF implementations.

C.1　Fortran 90 Features in the HPF 1.1 Subset

The features of the HPF 1.1 subset languages are listed below. For reference, the section numbers from the Fortran 90 standard are given along with the related syntax rule numbers:

- All FORTRAN 77 standard conforming features, except for storage and sequence association.
- The Fortran 90 definitions of MIL-STD-1753 features:
  - `DO WHILE` statement (8.1.4.1.1 / R821)
  - `END DO` statement (8.1.4.1.1 / R825)
  - `IMPLICIT NONE` statement (5.3 / R540)
  - `INCLUDE` line (3.4)
  - scalar bit manipulation intrinsic procedures: `IOR`, `IAND`, `NOT`, `IEOR`, `ISHFT`, `ISHFTC`, `BTEST`, `IBSET`, `IBCLR`, `IBITS`, `MVBITS` (13.13)
  - binary, octal and hexadecimal constants for use in `DATA` statements (4.3.1.1 / R407 and 5.2.9 / R533)
- Arithmetic and logical array features:
  - array sections (6.2.2.3 / R618–621)
    - subscript triplet notation (6.2.2.3.1)
    - vector-valued subscripts (6.2.2.3.2)
  - array constructors limited to one level of implied `DO` (4.5 / R431)
  - arithmetic and logical operations on whole arrays and array sections (2.4.3, 2.4.5, and 7.1)

  - array assignment (2.4.5, 7.5, 7.5.1.4, and 7.5.1.5)
  - masked array assignment (7.5.3)
    * `WHERE` statement (7.5.3 / R738)
    * block `WHERE` . . . `ELSEWHERE` construct (7.5.3 / R739)
  - array-valued external functions (12.5.2.2)
  - automatic arrays (5.1.2.4.1)
  - `ALLOCATABLE` arrays and the `ALLOCATE` and `DEALLOCATE` statements (5.1.2.4.3, 6.3.1 / R622, and 6.3.3 / R631)
  - assumed-shape arrays (5.1.2.4.2 / R516)

- Intrinsic procedures:

  The list of intrinsic functions and subroutines below is a combination of (a) routines which are entirely new to Fortran and (b) routines that have always been part of Fortran, but have been extended here to new argument and result types. The new or extended definitions of these routines are part of the subset. If a FORTRAN 77 routine is not included in this list, then only the original FORTRAN 77 definition is part of the subset.

  For all of the intrinsics that have an optional argument `DIM`, only actual argument expressions for `DIM` that are initialization expressions are part of the subset. The intrinsics with this constraint are marked with †in the list below.

  - the argument presence inquiry function: `PRESENT` (13.10.1)
  - all the numeric elemental functions: `ABS`, `AIMAG`, `AINT`, `ANINT`, `CEILING`, `CMPLX`, `CONJG`, `DBLE`, `DIM`, `DPROD`, `FLOOR`, `INT`, `MAX`, `MIN`, `MOD`, `MODULO`, `NINT`, `REAL`, `SIGN` (13.10.2)
  - all mathematical elemental functions: `ACOS`, `ASIN`, `ATAN`, `ATAN2`, `COS`, `COSH`, `EXP`, `LOG`, `LOG10`, `SIN`, `SINH`, `SQRT`, `TAN`, `TANH` (13.10.3)
  - all the bit manipulation elemental functions : `BTEST`, `IAND`, `IBCLR`, `IBITS`, `IBSET`, `IEOR`, `IOR`, `ISHFT`, `ISHFTC`, `NOT` (13.10.10)
  - all the vector and matrix multiply functions: `DOT_PRODUCT`, `MATMUL` (13.10.13)
  - all the array reduction functions: `ALL`†, `ANY`†, `COUNT`†, `MAXVAL`†, `MINVAL`†, `PRODUCT`†, `SUM`†(13.10.14)
  - all the array inquiry functions: `ALLOCATED`, `LBOUND`†, `SHAPE`, `SIZE`†, `UBOUND`†(13.10.15)
  - all the array construction functions: `MERGE`, `PACK`, `SPREAD`†, `UNPACK` (13.10.16)
  - the array reshape function: `RESHAPE` (13.10.17)
  - all the array manipulation functions: `CSHIFT`†, `EOSHIFT`†, `TRANSPOSE` (13.10.18)
  - all array location functions: `MAXLOC`†, `MINLOC`†(13.10.19)

- all intrinsic subroutines: DATE_AND_TIME, MVBITS, RANDOM_NUMBER, RANDOM_SEED, SYSTEM_CLOCK (3.11)

- Declarations:
  - Type declaration statements, with all forms of *type-spec* except *kind-selector* and TYPE(type-name), and all forms of *attr-spec* except *access-spec*, TARGET, and POINTER. (5.1 / R501-503, R510)
  - attribute specification statements: ALLOCATABLE, INTENT, OPTIONAL, PARAMETER, SAVE (5.2)

- Procedure features:
  - INTERFACE blocks with no *generic-spec* or *module-procedure-stmt* (12.3.2.1)
  - optional arguments (5.2.2)
  - keyword argument passing (12.4.1 /R1212)

- Syntax improvements:
  - long (31 character) names (3.2.2)
  - lower case letters (3.1.7)
  - use of "_" in names (3.1.3)
  - "!" initiated comments, both full line and trailing (3.3.2.1)

C.2 HPF 1.1 Directives and Language Extensions in the HPF 1.1 Subset

The following HPF 1.1 directives and language extensions to Fortran 90 were included in the HPF 1.1 Subset:

- The basic data distribution and alignment directives: ALIGN, DISTRIBUTE, PROCESSORS. and TEMPLATE.
- The *forall-statement* (but not the *forall-construct*).
- The INDEPENDENT directive.
- The SEQUENCE and NO SEQUENCE directives.
- The system inquiry intrinsic functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE.
- The computational intrinsic functions ILEN, and the HPF extended Fortran intrinsics MAXLOC and MINLOC, with the restriction that any actual argument expression corresponding to an optional DIM argument must be an initialization expression.

For a discussion of the rationale by which features were chosen for the HPF 1.1 Subset, please consult HPF Language Specification Version 1.1.

附属書D Previous HPFF Acknowledgments

The current HPF 2.0 document would not have been possible without the contributions of the previous series of HPFF meetings. Following are the acknowlegements for those efforts.

D.1 HPFF Acknowledgments

Technical development for HPF 1.0 was carried out by subgroups, and was reviewed by the full committee. Many people served in positions of responsibility:

- Ken Kennedy, Convener and Meeting Chair;
- Charles Koelbel, Executive Director and Head of the FORALL Subgroup;
- Mary Zosel, Head of the Fortran 90 and Storage Association Subgroup;
- Guy Steele, Head of the Data Distribution Subgroup;
- Rob Schreiber, Head of the Intrinsics Subgroup;
- Bob Knighten, Head of the Parallel I/O Subgroup;
- Marc Snir, Head of the Extrinsics Subgroup;
- Joel Williamson and Marina Chen, Heads of the Subroutine Interface Subgroup; and
- David Loveman, Editor.

Geoffrey Fox convened the first HPFF meeting with Ken Kennedy and later led a group to develop benchmarks for HPF. Clemens-August Thole organized a group in Europe and was instrumental in making this an international effort. Charles Koelbel produced detailed meeting minutes that were invaluable to subgroup heads in preparing successive revisions to the draft proposal. Guy Steele developed LaTeX macros for a variety of tasks, including formatting BNF grammar, Fortran code and pseudocode, and commentary material; the document would have been much less aesthetically pleasing without his efforts.

Many companies, universities, and other entities supported their employees' attendance at the HPFF meetings, both directly and indirectly. The following organizations were represented at two or more meetings by the following individuals (not including those present at the first HPFF meeting in January of 1992, for which there is no accurate attendee list):
Alliant Computer Systems Corporation .....David Reese

Amoco Production Company ..... Jerrold Wagener, Rex Page
Applied Parallel Research ..... John Levesque, Rony Sawdayi, Gene Wagenbreth
Archipel ..... Jean-Laurent Philippe
CONVEX Computer Corporation ..... Joel Williamson
Cornell Theory Center ..... David Presberg
Cray Research, Inc. ..... Tom MacDonald, Andy Meltzer
Digital Equipment Corporation ..... David Loveman
Fujitsu America ..... Siamak Hassanzadeh, Ken Muira
Fujitsu Laboratories ..... Hidetoshi Iwashita
GMD-I1.T, Sankt Augustin ..... Clemens-August Thole
Hewlett Packard ..... Maureen Hoffert, Tin-Fook Ngai, Richard Schooler
IBM ..... Alan Adamson, Randy Scarborough, Marc Snir, Kate Stewart
Institute for Computer Applications in Science & Engineering ... Piyush Mehrotra
Intel Supercomputer Systems Division ..... Bob Knighten
Lahey Computer ..... Lev Dyadkin, Richard Fuhler, Thomas Lahey, Matt Snyder
Lawrence Livermore National Laboratory ..... Mary Zosel
Los Alamos National Laboratory ..... Ralph Brickner, Margaret Simmons
Louisiana State University ..... J. Ramanujam
MasPar Computer Corporation ..... Richard Swift
Meiko, Inc. ..... James Cownie
nCUBE, Inc. ..... Barry Keane, Venkata Konda
Ohio State University ..... P. Sadayappan
Oregon Graduate Institute of Science and Technology ..... Robert Babb II
The Portland Group, Inc. ..... Vince Schuster
Research Institute for Advanced Computer Science ..... Robert Schreiber
Rice University ..... Ken Kennedy, Charles Koelbel
Schlumberger ..... Peter Highnam
Shell ..... Don Heller
State University of New York at Buffalo ..... Min-You Wu
SunPro and Sun Microsystems ..... Prakash Narayan, Douglas Walls
Syracuse University ..... Alok Choudhary, Tom Haupt
TNO-TU Delft ..... Edwin Paalvast, Henk Sips
Thinking Machines Corporation ..... Jim Bailey, Richard Shapiro, Guy Steele
United Technologies Corporation ..... Richard Shapiro
University of Stuttgart ..... Uwe Geuder, Bernhard Woerner, Roland Zink
University of Southampton ..... John Merlin
University of Vienna ..... Barbara Chapman, Hans Zima
Yale University ..... Marina Chen, Aloke Majumdar

Many people contributed sections to the final language specification and HPF Journal of Development, including Alok Choudhary, Geoffrey Fox, Tom Haupt, Maureen Hoffert, Ken Kennedy, Robert Knighten, Charles Koelbel, David Loveman, Piyush Mehrotra, John

Merlin, Tin-Fook Ngai, Rex Page, Sanjay Ranka, Robert Schreiber, Richard Shapiro, Marc Snir, Matt Snyder, Guy Steele, Richard Swift, Min-You Wu, and Mary Zosel. Many others contributed shorter passages and examples and corrected errors.

Because public input was encouraged on electronic mailing lists, it is impossible to identify all who contributed to discussions; the entire mailing list was over 500 names long. Following are some of the active participants in the HPFF process not mentioned above:

N. Arunasalam	Werner Assmann	Marc Baber
Babak Bagheri	Vasanth Bala	Jason Behm
Peter Belmont	Mike Bernhardt	Keith Bierman
Christian Bishof	John Bolstad	William Camp
Duane Carbon	Richard Carpenter	Brice Cassenti
Doreen Cheng	Mark Christon	Fabien Coelho
Robert Corbett	Bill Crutchfield	J. C. Diaz
James Demmel	Alan Egolf	Bo Einarsson
Pablo Elustondo	Robert Ferrell	Rhys Francis
Hans-Hermann Frese	Steve Goldhaber	Brent Gorda
Rick Gorton	Robert Halstead	Reinhard von Hanxleden
Hiroki Honda	Carol Hoover	Steven Huss-Lederman
Ken Jacobsen	Elaine Jacobson	Behm Jason
Alan Karp	Ronan Keryell	Anthony Kimball
Ross Knippe	Bruce Knobe	David Kotz
Ed Krall	Tom Lake	Peter Lawrence
Bryan Lawver	Bruce Leasure	Stewart Levin
David Levine	Theodore Lewis	Woody Lichtenstein
Ruth Lovely	Doug MacDonald	Raymond Man
Stephen Mark	Philippe Marquet	Jeanne Martin
Oliver McBryan	Charlie McDowell	Michael Metcalf
Charles Mosher	Len Moss	Lenore Mullin
Yoichi Muraoka	Bernie Murray	Vicki Newton
Dale Nielsen	Kayutov Nikolay	Steve O'Neale
Jeff Painter	Cherri Pancake	Harvey Richardson
Bob Riley	Kevin Robert	Ron Schmucker
J.L. Schonfelder	Doug Scofield	David Serafini
G.M. Sigut	Anthony Skjellum	Niraj Srivastava
Paul St.Pierre	Nick Stanford	Mia Stephens
Jaspal Subhlok	Xiaobai Sun	Hanna Szoke
Bernard Tourancheau	Anna Tsao	Alex Vasilevsky
Stephen Vavasis	Arthur Veen	Brian Wake
Ji Wang	Karen Warren	D.C.B. Watson
Matthijs van Waveren	Robert Weaver	Fred Webb
Stephen Whitley	Michael Wolfe	Fujio Yamamoto
Marco Zagha		

The following organizations made the language draft available by anonymous FTP access and/or mail servers: AT&T Bell Laboratories, Cornell Theory Center, GMD-I1.T (Sankt Augustin), Oak Ridge National Laboratory, Rice University, Syracuse University, and Thinking Machines Corporation. These outlets were instrumental in distributing the document.

The High Performance Fortran Forum also received a great deal of volunteer effort in nontechnical areas. Theresa Chatman and Ann Redelfs were responsible for most of the meeting planning and organization, including the first HPFF meeting, which drew over 125 people. Shaun Bonton, Rachele Harless, Rhonda Perales, Seryu Patel, and Daniel Swint helped with many logistical details. Danny Powell spent a great deal of time handling the financial details of the project. Without these people, it is unlikely that HPF would have been completed.

HPFF operated on a very tight budget (in reality, it had no budget when the first meeting was announced). The first meeting in Houston was entirely financed from the conferences budget of the Center for Research on Parallel Computation, an NSF Science and Technology Center. DARPA and NSF have supported research at various institutions that have made a significant contribution towards the development of High Performance Fortran. Their sponsored projects at Rice, Syracuse, and Yale Universities were particularly influential in the HPFF process. Support for several European participants was provided by ESPRIT through projects P6643 (PPPE) and P6516 (PREPARE).

D.2 HPFF94 Acknowledgments

The HPF 1.1 version of the document was prepared during the HPFF94 series of meetings. A number of people shared technical responsibilities for the activities of the HPFF94 meetings:

- Ken Kennedy, Convener and Meeting Chair;
- Mary Zosel, Executive Director and head of CCI Group 2;
- Richard Shapiro, Head of CCI Group 1;
- Ian Foster, Head of Tasking Subgroup;
- Alok Choudhary, Head of Parallel I/O Subgroup;
- Chuck Koelbel, Head of Irregular Distributions Subgroup;
- Rob Schreiber, Head of Implementation Subgroup;
- Joel Saltz, Head of Benchmarks Subgroup;
- David Loveman, Editor, assisted by Chuck Koelbel, Rob Schreiber, Guy Steele, and Mary Zosel, section editors.

Attendence at the HPFF94 meetings included the following people from organizations that were represented two or more times.

Don Heller ..... Ames Laboratory
Jerrold Wagener ..... Amoco Production Company
John Levesque ..... Applied Parallel Research
Ian Foster ..... Argonne National Laboratory
Terry Pratt ..... CESDIS/NASA Goddard
Jim Cowie ..... Cooperating Systems
Andy Meltzer, Jon Steidel ..... Cray Research, Inc.
David Loveman ..... Digital Equipment Corporation
Bruce Olsen ..... Hewlett Packard
E. Nunohiro, Satoshi Itoh ..... Hitachi
Henry Zongaro ..... IBM
Piyush Mehrotra ... Institute for Computer Applications in Science & Engineering
Bob Knighten, Roy Touzeau ..... Intel SSD
Mary Zosel, Bor Chan, Karen Warren ... Lawrence Livermore National Laboratory
Ralph Brickner ..... Los Alamos National Laboratory
J. Ramanujam ..... Louisiana State University
Paula Vaughan, Donna Reese ..... Mississippi State University and NSF ERC
Shoichi Sakon, Yoshiki Seo ..... NEC
P. Sadayappan, Chua-Huang Huang ..... Ohio State University
Andrew Johnson ..... OSF Research Institute
Chip Rodden, Jeff Vanderlip ..... Pacific Sierra Research
Larry Meadows, Doug Miles ..... The Portland Group, Inc.
Robert Schreiber ..... Research Institute for Advanced Computer Science
Ken Kennedy, Charles Koelbel ..... Rice University
Ira Baxter ..... Schlumberger
Alok Choudhary ..... Syracuse University
Guy Steele ..... Thinking Machines Corporation, Sun Microsystems
Richard Shapiro ..... Thinking Machines Corp., Silicon Graphics Inc.
Scott Baden, Val Donaldson ..... University of California, San Diego
Robert Babb ..... University of Denver
Joel Saltz, Paul Havlak ..... University of Maryland
Nicole Nemer-Preece ..... University of Missouri-Rolla
Hans Zima, Siegfried Benkner, Thomas Fahringer ..... University of Vienna

An important activity of HPFF94 was the processing of the many items submitted for comment and interpretation which led to the HPF 1.1 update of the language document. A special acknowlegement goes to Henry Zongaro, IBM, for many thoughtful questions exposing dark corners of language design that were previously overlooked, and to Guy Steele, Thinking Machines/Sun Microsystems for his analysis of, and solutions for some of the thornier issues discussed. And general thanks to the people who submitted comments and interpretation requests, including:

David Loveman, Michael Hennecke, James Cownie, Adam Marshall, Stephen Ehrlich, Mary Zosel, Matt Snyder, Larry Meadows, Dick Hendrickson, Dave Watson, John Merlin, Vasanth Bala, Paul.Wesson, Denis.Hugli, Stanly Steinberg, Henk Sips, Henry Zongaro, Eiji_Nunohiro, Jens Bloch Helmers, Rob Schreiber, David B. Serafini, and Allan Knies.

Other special mention goes to Chuck Koelbel at Rice University for continued maintenance of the HPFF mailing lists, to Donna Reese and staff at Mississippi State University for establishing and maintaining a WWW home-page for HPFF, and to the University of Maryland for establishing a benchmark FTP site.

Theresa Chatman and staff at Rice University were responsible for meeting planning and organization and Danny Powell continued to handle financial details of the project.

HPFF94 received direct support for research and administrative activities from grants from ARPA, DOE, and NSF.

関連図書

[1] American National Standards Institute, Inc., 1430 Broadway, New York, NY. *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, approved April 3, 1978.

[2] American National Standards Institute, Inc., 1430 Broadway, New York, NY. *American National Standard for Information Systems Programming Language FORTRAN*, S8 (X3.9-198x) Revision of X3.9-1978, Draft S8, Version 104, April 1987.

[3] High Performance Fortran Forum. *High Performance Fortran Language Specification* Scientific Programming, 2,1, 1993. Also published as: CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised May. 1993). Also published as: Fortran Forum, 12,4, Dec. 1993 and 13,2, June 1994.

[4] High Performance Fortran Forum. *High Performance Fortran Language Specification, version 1.0* CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, 1992 (revised May. 1993).

[5] US Department of Defense. *Military Standard, MIL-STD-1753: FORTRAN, DoD Supplement to American National Standard X3.9-1978*, November 9, 1978.

附属書E Policy and Mechanism for Recognized Extrinsic Interfaces

HPF defines certain extrinsics such as `HPF_LOCAL`, and `HPF_SERIAL` as interfaces that HPFF believes are useful to the HPF community. But there are many more such extrinsic interfaces beyond those maintained by HPFF. HPFF has a adopted a policy of formally recognizing certain extrinsic interface definitions, where the interface, and its addition to the HPF document is considered to be a service to the HPF community. Examples are language bindings to HPF or library packages.

E.1 Extrinsic Policy

To be considered for HPFF recognition, a proposed extrinsic must demonstrate the following things. It should be noted, however, that meeting these criteria does not guarentee acceptance of a proposed interface by HPFF.

- conformance to HPF rules for calling extrinsics,
- significant new functionality,
- existing practice such as users, implementations, etc.,
- institutional backing with evidence of ongoing support,
- coherent documentation,
- non-proprietary interface definition, and
- copyright goes to HPFF for interface, with permission to use (royalty free).

If a proposed extrinsic is accepted by HPFF, then:

- HPFF will recogize the interface and reference it in documentation, but HPFF does not assume responsibility for the extrinsic or its interface.
- The sponsor of the extrinsic must continue to conform to the HPF interface rules for extrinsics. The interface HPFF approves must not change without HPFF approval.
- The sponsor must assume responsibility for any CCI requests concernting the extrinsic.

A list of recognized extrinsic interfaces will be included in HPF documentation, with the following guidelines:

- There should be a single page introduction to the extrinsic which contains:
  - the name of the extrinsic,
  - a brief abstract of functionality,
  - a brief and informal description of the interface,
  - information about platform and system availability, and
  - reference and contacts for formal documentation, continued responsibility, and additional information (e.g. compiler availability).
- There should be about two pages with short examples of usage.
- A short paper with the formal definition of the interface and an informal description of the functionality of the extrinsic.

E.2 Extrinsic Interface Mechanism

The HPF www-home page will have instructions for submission of an extrinsic interface. For HPFF consideration, the sponsor prepares a proposal that includes:

- a statement of what significant new functionality is provided,
- a description of existing practice,
- a statement of institutional backing with evidence of ongoing support,
- a copy of the complete documentation or a reference to an online version of the documentation,
- a draft of the text (described above) that would be included in the HPFF documentation, and
- a statement justifying the claim that the interface follows HPF conventions for calling extrinsics.

If the proposed extrinsic interface is approved by HPFF, the sponsor then submits:

- a formal statement for HPFF records that the interface definition is non-proprietary and that the copyright of the interface belongs HPFF,
- the formal contact for CCI and continued maintenance of the interface, and
- a copy of the interface documentation formatted for HPFF use, including a copy in the current document and web mark-up languages.

附属書F　HPF_CRAFT

HPF_CRAFT is a hybrid language, combining an SPMD execution model with high performing HPF features. The model combines the multi-threaded execution of HPF_LOCAL and the HPF syntax. The goal of HPF_CRAFT is to attain the potential performance of an SPMD programming model with access to HPF features and a well-defined extrinsic interface to HPF.

F.1 Introduction

HPF_CRAFT is a hybrid language, combining an SPMD execution model with high performing and portable HPF features. The model combines the multi-threaded execution of HPF_LOCAL and the HPF syntax and features. The goal of HPF_CRAFT is to attain the potential performance of an SPMD programming model with access to HPF features and a well-defined extrinsic interface to HPF. It is built on top of the HPF_LOCAL extrinsic environment.

SPMD features and a multi-threaded model allow the user to take advantage of the performance and opportunity for low level access of a more general purpose programming model. Including HPF data distribution features gives the programmer access to high performing aspects of both models, but with the added responsibility of working with a more low-level execution model. HPF_CRAFT is best suited for platforms that support one way communication features, but is consistent with HPF and easily targeted for platforms that have HPF and can support SPMD programming styles.

The HPF features included in HPF_CRAFT are a subset of the full HPF language chosen for their performance and their broad portability and ease of use. HPF_CRAFT contains additional features to support SPMD programming styles. There are some differences from HPF, however. For example, I/O causes differences; in HPF_CRAFT different processors are allowed to read from different files at the same time, in HPF the processors must all read from the same file. The differences in the models are principally caused by the multi-threaded execution model and the introduction of HPF_LOCAL data rules.

HPF_CRAFT allows for the notion of *private data*. Data defaults to a mapping in which data items are allocated so that each processor has a unique copy. The values of the individual data items and the flow of control may vary from processor to processor within HPF_CRAFT. This behavior is consistent with the behavior of HPF_LOCAL. In HPF_CRAFT a processor may be individually named and code executed based upon which processor it is executing on. HPF_CRAFT also allows for the notion of *private loops*. A private loop is executed in entirety by each processor.

The rules governing the interface to HPF_CRAFT subprograms are similar to those for the HPF_LOCAL interface. Dummy arguments use a hybrid of the interfaces between HPF and itself and that of HPF and HPF_LOCAL. Explicitly mapped dummy arguments behave just as they do in HPF, while default (private) dummy arguments use the HPF_LOCAL calling convention.

HPF_CRAFT will be initially made available on Cray MPP systems and may also be available on Cray vector architectures. Future versions of HPF_CRAFT are possible on other vendor's architectures as well.

HPF_CRAFT is being implemented for Cray Research by The Portland Group, Inc. For Cray systems, HPF_CRAFT may be obtained through the Cray Research Inc. Orderdesk,

Cray Research Inc.
orderdsk@cray.com
(612) 683-5907

Additional formal documentation, requests, and suggestions can be made to

The Portland Group
9150 SW Pioneer Ct., Suite H
Wilsonville, OR 97070
(503) 682-2806
trs@pgroup.com

F.2 Examples of Use

HPF_CRAFT is intended for use in circumstances where greater control and performance are desired for MIMD style architectures. Since data may be declared to be private, local control is made more available and since processor information is available message passing and direct memory access programming styles can be seamlessly integrated with explicitly mapped data.

The following examples show some of the capabilities of HPF_CRAFT that are different from those of HPF. Others, such as integrated message passing and synchronization primitives are not shown. Much of HPF can also be used within HPF_CRAFT.

Example 1 illustrates the difference between the default distribution for data and the distribution of mapped data.

```
!  Example 1

       INTEGER PRIVATE_A(100, 20), PRIVATE_B(12, 256), PRIVATE_C
       INTEGER MAPPED_A(100, 20), MAPPED_B(12, 256), MAPPED_C
!HPF$ DISTRIBUTE MAPPED_A(BLOCK, BLOCK), MAPPED_B(BLOCK, *), MAPPED_C
```

In the above example, given 8 processors, there would be 8 * 100 * 20 (or 16,000) elements in the array PRIVATE_A. Each processor contains an entire array named PRIVATE_A. The elements of PRIVATE_A on processor 1 cannot be referenced using implicit syntax by any other processor. There are only 100 * 20 (or 2000) elements of array MAPPED_A, however, and these elements are distributed about the machine in a (BLOCK, BLOCK) fashion.

The difference between the PRIVATE_A declaration in HPF_CRAFT and that in HPF is the most instructive. In HPF_CRAFT each processor contains one copy of the array, and the values of the elements of the array may vary from processor to processor. HPF implementations are permitted to make one copy of the array per processor the default, but the values of these copies must remain coherent across all processors. In HPF there is no way to write a conforming program in which different processors have different values for the same array.

Example 2 shows the usefulness of the ON clause for the INDEPENDENT loop as well as giving an example of how private data may be used.

```
!  Example 2

       PRIVATE_C = 0
!HPF$  INDEPENDENT (I, J) ON MAPPED_B(I, J)
       DO J=1,256
          DO I=1,12
             MAPPED_B(I, J) = MAPPED_B(I, J) + 5
             PRIVATE_C = PRIVATE_C + MAPPED_B(I, J)
          ENDDO
       ENDDO
```

In this example, each iteration is executed on the processor containing the data that is mapped to it. The user was allowed to specify this.

In addition, the private variable PRIVATE_C is used to compute a total for each processor. At the end of execution of the loop, the values of PRIVATE_C may be different on each processor depending upon the values in the elements of the array on each processor. This data may be used as is, or it can be quickly summed using a barrier or an ATOMIC UPDATE.

Example 3 shows the final total value being combined into the variable MAPPED_C whose value is available to all processors.

```
!  Example 3

       MAPPED_C = 0
!HPF$  ATOMIC UPDATE
       MAPPED_C = MAPPED_C + PRIVATE_C
```

Example 4 shows how the language allows private data to vary from processor to processor.

```
!  Example 4

       IF (MY_PE() .EQ. 5) THEN
          PRIVATE_C =  some-big-expression
       ENDIF
```

In this example, PRIVATE_C on processor 5 will have the result of *some-big-expression*. Each processor can do distinctly different work and communicate through mapped data.

The code fragment in Example 5 is from an application and shows a few features of the language.

```
!  Example 5

!HPF$ GEOMETRY G(*, CYCLIC)
      REAL FX(100,100), FY(100,100), FZ(100,100)
!HPF$ DISTRIBUTE (G) ::  FX,FY,FZ
      REAL FXP(100,16,100), FYP(100,16,100)
!HPF$ DISTRIBUTE FXP(*,*, BLOCK) FYP(*,*, BLOCK)
      INTEGER CELL, ATOM, MAP(1000), NACELL(1000)

!HPF$ INDEPENDENT (CELL) ON FX(1,CELL)
      DO CELL=1,100
         JCELL0 = 16*(CELL-1)
         DO NABOR = 1, 13
            JCELL = MAP(JCELL0+NABOR)
            DO ATOM=1, NACELL(CELL)
               FX(ATOM, CELL) = FX(ATOM, CELL) + FXP(ATOM, NABOR, JCELL)
               FY(ATOM, CELL) = FY(ATOM, CELL) + FYP(ATOM, NABOR, JCELL)
            ENDDO
         ENDDO
      ENDDO
```

The GEOMETRY directive allows the user to generically specify a mapping and use it to apply to many arrays (they need not have the same extents.)

Example 5 has a single INDEPENDENT loop which is the outer loop. It executes 100 iterations total. Within this loop the private value of JCELL0 is set for each processor (ensuring that it is a local computation everywhere.) Nested inside the INDEPENDENT loop is a private loop; this loop executes 13 times *per* processor. Inside this loop JCELL is

computed locally on each processor, minimizing unnecessary communication. Finally the innermost loop is also private.

F.3 External Interface

This section describes the behavior when an HPF_CRAFT routine is called from HPF.

The calling convention and argument passing rules for HPF_CRAFT are a hybrid of those for HPF calling HPF_LOCAL and HPF calling HPF. Explicit interfaces are required. Where dummy arguments are private (default) storage, the HPF calling HPF_LOCAL conventions are used. Where dummy arguments are explicitly mapped, the calling convention matches HPF calling HPF.

There are a number of constraints on HPF_CRAFT routines that are called from HPF. The following is a list of restrictions placed on HPF_CRAFT routines called from HPF:

- Recursive HPF_CRAFT routines cannot be called from HPF.
- HPF_CRAFT routines called from HPF may only enter the routine at a single place (no alternate entries).
- An HPF_CRAFT supprogram may not be invoked directly or indirectly from within the body of a `FORALL` construct or within the body of an `INDEPENDENT DO` loop that is inside an HPF program.
- The attributes (type, kind, rank, optional, intent) of the dummy arguments in a supprogram called by HPF must match the attributes of the corresponding dummy arguments in the explicit interface.
- A dummy argument of an HPF_CRAFT supprogram called by HPF
  - must not be a procedure name.
  - must not have the `POINTER` attribute.
  - must not be sequential, unless it is also `PE_PRIVATE`.
  - must have assumed shape even when it is explicit shape in the interface.
  - if scalar, it must be mapped so that each processor has a copy of the argument.
- The default mapping of scalar dummy arguments and of scalar function results when an HPF program calls an HPF_CRAFT routine is that it is replicated on each processor.

If a dummy argument of an `EXTRINSIC('HPF_CRAFT')` routine interface block is an array and the dummy argument of the HPF_CRAFT supprogram has the default private mapping, then the corresponding dummy argument in the specification of the HPF_CRAFT procedure must be an array of the same rank, type, and type parameters. When the extrinsic procedure is invoked, the dummy argument is associated with the local array that consists of the subgrid of the global array that is stored locally.

If the dummy argument of the HPF_CRAFT supprogram is explicitly mapped, it must have the same mapping as the dummy argument of the `EXTRINSIC('HPF_CRAFT')` supprogram. Note that this restriction does not require actual and dummy arguments to match and is no more stringent than saying that mappings of dummy arguments in interface blocks must match those in the actual routine.

F.4 Execution Model

HPF_CRAFT is built upon the fundamental execution model of HPF_LOCAL, augmented with data mapping and work distribution features from HPF. It is also augmented with explicit low-level control features, many taken from Cray Research's CRAFT language.

In HPF_CRAFT there is a single task on each processor and all tasks begin executing in parallel, with data defaulting to a private distribution, the same default distribution used in HPF_LOCAL. Each processor gets a copy of the data storage unless specified otherwise by the user. Consequently I/O works identically to I/O in HPF_LOCAL and message passing libraries are easily integrated.

Simply stated, the execution model is that of HPF_LOCAL.

To provide correct behavior when explicitly mapped data is involved, this model defines implicit barrier points at which the execution model requires that all processors must stop and wait for the execution of all other processors before continuing. These barriers add additional semantics to the HPF_LOCAL behavior. An implementation may remove any of these barriers that are deemed unnecessary, but *every* processor must participate in the barriers at each one of these points.

The points where there are implicit barriers are conceptually after those instances in which the processors in the HPF_CRAFT program are executing cooperatively, as if in an HPF program (e.g., after an `INDEPENDENT` loop). An HPF_CRAFT program treats operations on explicitly mapped objects as if they were operations in an HPF program and it treates operations on private data as if they were executed within the HPF_LOCAL framework. It is occasionally useful for an advanced programmer to indicate to the compilation system where barriers are not needed; HPF_CRAFT has syntax to allow this capability.

F.5 HPF_CRAFT Functional Summary

HPF_CRAFT contains a number of features not available in HPF, and restricts the usage of many of the features currently available. The following is a concise list of the differences.

- `INDEPENDENT` has been extended to better support an `ON` clause.
- There are new rules defining the interaction of explicitly mapped and private data.
- Parallel inquiry intrinsics `IN_PARALLEL()` and `IN_INDEPENDENT()` have been added.
- Serial regions (`MASTER / END MASTER`) have been added.
- Explicit synchronization primitives are provided.

- The `ATOMIC UPDATE`, `SYMMETRIC`, and `GEOMETRY` directives have been added.
- Many other compiler information directives have been added to assist the compiler in producing good quality code.

F.5.1 Data Mapping Features

Data mapping features provided are those that have been found useful most often. When data is explicitly mapped, only one copy of the data storage is created unless the explicit mapping directs otherwise. The value of explicitly mapped replicated data items must be consistent between processors as is the case in HPF. Storage and sequence association for explicitly mapped arrays is not guaranteed in HPF_CRAFT. For private data, storage and sequence association follows the Fortran 90 rules.

A new directive is included for completeness: `PE_PRIVATE`, which specifies that the data should conform to the default behavior. The values of private varaibles may vary on different processors.

F.5.2 Subprogram Interfaces

The behavior and requirements of an HPF_CRAFT program at subprogram interfaces may be divided into three cases. Each case is also available using some combination of HPF and HPF_LOCAL. For dummy arguments that are explicitly mapped, the behavior is identical to that of HPF. All processors must cooperate in a subprogram invocation that remaps or explicitly maps data. In other words, if an explicit interface is required (by the HPF rules) or the subprogram declares explicitly mapped data, the subprogram must be called on all processors. Processors need not cooperate if there are only reads to non-local data. The `INHERIT` attribute may only be applied to explicitly mapped data.

Data that has the default private mapping (case two) the behavior of an HPF_CRAFT subprogram at subprogram interfaces is identical to that of HPF_LOCAL. Data is passed individually on every processor and the processors need not interact in any way.

When a subprogram is passed actual arguments that are a combination of both explicitly mapped data and private data, the explicitly mapped data follows the HPF rules and the private data follows the HPF_LOCAL rules.

In case three, the user has the option of passing data with explicitly mapped actual arguments to dummy arguments that are not explicitly mapped (i.e., private.) The mapping rules for this data are identical to the mapping rules when HPF calls an HPF_LOCAL subprogram. The data remains "in-place." All HPF arrays are logically carved up into pieces; the HPF_CRAFT procedure executing on a particular physical processor sees an array containing just those elements of the global array that are mapped to that physical processor. There is implicit barrier synchronization after an `INDEPENDENT` loop. Transfer of control into or out of an `INDEPENDENT` loop is prohibited.

Finally, it is undefined behavior when an actual argument is private and the dummy argument is explicitly mapped. A definition could be supplied for this interaction, but

it is the same solution that one might propose for a calling sequence when HPF_LOCAL subprograms call HPF subprograms.

F.5.3 The INDEPENDENT Directive

The `INDEPENDENT` directive is part of HPF_CRAFT with the same semantics as in HPF. However, within `INDEPENDENT` loops the values of private data may vary from processor to processor. `INDEPENDENT` applied to `FORALL` has identical syntax and semantics as in HPF.

An HPF independent loop optionally may have a `NEW` clause. The `NEW` clause is not required by HPF_CRAFT for default (not explicitly mapped) data. In HPF_CRAFT data defaults to private so values may differ from processor to processor.

Private data has slightly different behavior than data specified in the `NEW` clause. The value of a private datum on each processor can be used beyond a single iteration of the loop. Private data may be used to compute local sums, for example. The values of data items named in a `NEW` clause may not be used beyond a single iteration. The `NEW` clause asserts that the `INDEPENDENT` directive would be valid if new objects were created for the variables named in the clause for each iteration of the loop. The semantics of the `NEW` clause are identical in HPF_CRAFT and HPF.

The semantics of an `INDEPENDENT` applied to loops containing private data references changes with respect to the private data. The change can be summarized to say that instead of indicating that iterations have no dependencies upon one-another, with respect to the private data, iterations on different processors have no dependencies upon one-another.

F.5.4 The ON Clause

In addition to the version of `INDEPENDENT` available from HPF, a new version of `INDEPENDENT` is included that incorporates the `ON` clause. There are a number of differences between the versions of `INDEPENDENT` with and without the `ON` clause.

The new version of the `INDEPENDENT` directive may be applied to the first of a group of tightly nested loops and may apply to more than one of them. This more easily facilitates the use of the `ON` clause. The current `INDEPENDENT` directive applies only to a single loop nest. The `INDEPENDENT` directive is extended so that multiple loop nests can be named. The general syntax for these new independent loops is as follows:

```
!HPF$ INDEPENDENT (I1,I2,...,In) ON array-name(h1(I1),h2(I2),...,hn(In))
      DO I1 = L1, U1, S1
            DO I2 = L2, U2, S2
              ...
                DO In = Ln, Un, Sn
                  ...
                END DO
              ...
            END DO
      END DO
```

The syntax and semantics of `INDEPENDENT` with the `ON` clause are different from its syntax and semantics without the `ON` clause. With the `ON` clause the directive states that there are no cross-processor dependencies, but there may be dependencies between iterations on a processor. There is an implicit barrier synchronization after an `INDEPENDENT` loop. Transfer of control into or out of an `INDEPENDENT` loop is prohibited.

The iteration space of an `INDEPENDENT` nest must be rectangular. That is, the lower loop bound, the upper loop bound, and the step expression for each loop indicated by the `INDEPENDENT` induction list must be invariant with regard to the `INDEPENDENT` nest. Each index expression of *array-name* in the `ON` clause (the functions h_i above,) must be one of the following two forms:

`[` *a* `* loop_control_variable +` `]` *b*
`[` *a* `* loop_control_variable -` `]` *b*

where *a* and *b* must be integer values; they can be expressions, constants, or variables. The values of *a* and *b* must be invariant with regard to the `INDEPENDENT` loop nest. For example, specifying `A(I,J,K)` is valid. Specifying `A(3,I+J,K)` is not valid. Specifying `A(I,I,K)` is not valid because `I` appears twice. Division is prohibited in any index expression of the `ON` clause.

F.5.5 Array Syntax

Array syntax is treated identically in HPF_CRAFT as in HPF for explicitly mapped objects. For private objects the behavior is identical to that of HPF_LOCAL. When private objects and explicitly mapped objects are combined the rules are as follows:

$$result = rhs_1 \ \mathrm{op}_1 \ rhs_2 \ \mathrm{op}_2 \ \ldots \ \mathrm{op}_m \ rhs_n$$

- If *result* is explicitly mapped and all *rhs* arrays are explicitly mapped, the work is distributed as in HPF.
- If *result* is private and all *rhs* arrays are private the computation is done on all processors as an HPF_LOCAL program would do it.
- If *result* is private and all *rhs* arrays are explicitly mapped, the work is distributed as in HPF and the values of the results are broadcast to the *result* on each processor.
- If *result* is explicitly mapped and *not* all *rhs* arrays are explicitly mapped, the results of the operation are undefined, unless all corresponding elements of all private *rhs* arrays have the same values.
- If *result* is private and some, but not all *rhs* arrays are explicitly mapped, the value is computed on each processor and saved to the local *result*.

All processors must participate in any array syntax statement in which the value of an explicitly mapped array is modified, and there is implicit barrier synchronization after the statement executes.

F.5.6 Treatment of FORALL and WHERE Statements

The FORALL and WHERE statements are treated exactly as in HPF when data is explicitly mapped. When private data is modified, the statement is executed separately on each processor. Finally, when data in a FORALL or WHERE are mixed, the rules for array syntax apply. If any explicitly mapped data item is modified in a *forall-stmt* or *where-stmt* then arrays in the *forall-header* or *where-header* must be explicitly mapped. In a FORALL construct, if any explicitly mapped array is modified, all modified arrays must be explicitly mapped. There is an implicit barrier synchronization after FORALL and WHERE statements if any arrays in the *forall-header* or *where-header* are explicitly mapped.

F.5.7 Synchronization Primitives

A number of synchronization primitives are provided. These primitives include:

Barriers (test, set, wait)
Locks (test, set, clear)
Critical Sections
Events (test, set, wait, clear)

Barriers provides an explicit mechanism for a task to indicate its arrival at a program point and to wait there until all other tasks arrive. A task may test and optionally wait at an explicit barrier point. In the following example, a barrier is used to make sure that *block3* is not entered by any task until all tasks have completed execution of *block1*.

```
block1
CALL SET_BARRIER()
block2
CALL WAIT_BARRIER()
block3
```

The following example performs a similar function as above. However, while waiting for all tasks to arrive at the barrier, the early tasks perform work within a loop.

```
block1
CALL SET_BARRIER()
DO WHILE (.NOT. TEST_BARRIER())
      block2
END DO
block3
```

Locks are used to prevent the simultaneous access of data by multiple tasks.

The SET_LOCK(*lock*) intrinsic sets the mapped integer variable *lock* atomically. If the lock is already set, the task that called SET_LOCK is suspended until the lock is cleared by another task and then sets it. Individual locks may be tested or cleared using *result* = TEST_LOCK(*lock*) and CLEAR_LOCK(*lock*) respectively.

A *critical section* protects access to a section of code rather than to a data object. The `CRITICAL` directive marks the beginning of a code region in which only one task can enter at a time. The `END CRITICAL` directive marks the end of the critical section. Transfer of control into or out of a critical section is prohibited.

```
!HPF$ CRITICAL
      GLOBAL_SUM = GLOBAL_SUM + LOCAL_SUM
!HPF$ END CRITICAL
```

Events are typically used to record the state of a program's execution and to communicate that state to another task. Because they do not set locks, as do the lock routines described earlier, they cannot easily be used to enforce serial access of data. They are suited to work such as signalling other tasks when a certain value has been located in a search procedure. There are four routines needed to perform the event functions, and each requires a mapped argument.

The `SET_EVENT(`*event*`)` routine sets or *posts* an event; it declares that an action has been accomplished or a certain point in the program has been reached. A task can post an event at any time, whether the state of the event is cleared or already posted. The `CLEAR_EVENT(`*event*`)` routine clears an event, the `WAIT_EVENT(`*event*`)` routine waits until a particualr event is posted, and the *result* = `TEST_EVENT(`*event*`)` function returns a logical value indicating whether a particular event has been posted.

F.5.8 Barrier Removal

You can explicitly remove an implicit barrier after any `INDEPENDENT` loop, or after any array syntax statement that modifies explicitly mapped arrays, by using the `NO BARRIER` directive.

```
!HPF$ NO BARRIER
```

F.5.9 Serial Regions

It is often useful to enter a region where only one task is executing. This is particularly useful for certain types of I/O. To facilitate this, two directives are provided. In addition, one may optionally attach a `COPY` clause to the `END MASTER` directive which specifies the private data items whose values should be broadcast to all processors. The syntax of this directive is:

```
!HPF$ MASTER
      sequential region
      ...
!HPF$ END MASTER [, COPY( var_1 [, var_2, ..., var_n ])]
```

where *var* is SYMMETRIC private data to be copied to the same named private data on other processors.

If a routine is called within a serial region, the routine executes serially; there is no way to get back to parallel execution within the routine. All explicitly mapped data is accessible from within routines called in a serial region, but a routine called from within a serial region cannot allocate explicitly mapped data or remap data. All processors must participate in the invocation of the serial region. Transfer of control into or out of a serial region is not permitted.

F.5.10 Libraries

The HPF Local Routine Library is available in HPF_CRAFT. The HPF_LOCAL extrinsic environment contains a number of libraries that are useful for local SPMD programming and a number of libraries that allow the user to determine global (rather than local) state information. These library procedures take as input the name of a dummy argument and return information on the corresponding global HPF actual argument. They may only be invoked by an HPF_CRAFT procedure that was directly invoked by global HPF code. They may be called only for private data. The libraries reside in a module called HPF_LOCAL_LIBRARY.

The HPF Library is available to HPF_CRAFT when called with data that is explicitly mapped and all processors are participating in the call. In addition, as in HPF_LOCAL, the entire HPF Library is available for use with private data. Mixing private and explicitly mapped data in calls to the HPF library produces undefined behavior.

F.5.11 Parallel Inquiry Intrinsics

These intrinsic functions are provided as an extension to HPF. They return a logical value that provides information to the programmer about the state of execution in a program.

```
IN_PARALLEL()
IN_INDEPENDENT()
```

F.5.12 Task Identity

MY_PE() may be used to return the local processor number. The physical processors are identified by an integer in the range of 0 to *n-1* where *n* is the value returned by the global HPF_LIBRARY function NUMBER_OF_PROCESSORS. Processor identifiers are returned by ABSTRACT_TO_PHYSICAL, which establishes the one-to-one correspondence between the abstract processors of an HPF processors arrangement and the physical processors. Also, the local library function MY_PROCESSOR returns the identifier of the task executing the call.

F.5.13 Parallelism Specification Directives

These directives allow a user to assert that a routine will only be called from within a parallel region, a serial region, or from within both regions. Without these directives an

implementation might be required to generate two versions of code for each routine, depending upon implementation strategies. The directives simply make the generated code size smaller and remove a test.

```
!HPF$ PARALLEL_ONLY
!HPF$ SERIAL_ONLY
!HPF$ PARALLEL_AND_SERIAL
```

The default is PARALLEL_ONLY.

F.5.14 The SYMMETRIC Directive

SYMMETRIC variables are private data that are guaranteed to be at the same storage location on every processor. The feature is beneficial to implementations that provide one-way communication functionality. One task can either *get* or *put* data into another task's symmetric data location, without involving the other task. There is an implicit barrier synchronization after SYMMETRIC data is allocated.

```
      REAL PRIV1(100), PRIV2
!HPF$ SYMMETRIC PRIV1, PRIV2
```

F.5.15 The RESIDENT Directive

The RESIDENT directive can be specified at the loop level and at the routine level. It is an assertion that the references to particular variables in the routine (or loop) are only references to data that are local to the task making the assertion. In the following loop, all references to arrays A, B, and C are local to the task executing each iteration.

```
      REAL A(100), B(100), C(100)
      INTEGER IX(100)
!HPF$ DISTRIBUTE A(BLOCK), B(BLOCK), C(BLOCK)
!HPF$ RESIDENT A

      ...
!HPF$ INDEPENDENT (I) ON B(I) RESIDENT(C)
      DO I = 1, 100
            A(IX(I)) = B(I) + C(IX(I))
      END DO
```

F.5.16 The ATOMIC UPDATE Directive

In HPF_CRAFT, the ATOMIC UPDATE directive tells the compiler that a particular data item or the elements of a particular array for a specified operation must be updated atomically. This can be used within loops or in array syntax and applies to both the elements of an array with an assignment of a permutation and the elements of an array within a loop.

In the following example, all references to `R(IX(I))` occur atomically, thus eliminating the possibility that different iterations might try to modify the same element concurently.

```
      REAL R(200), S(1000)
      INTEGER IX(1000)
!HPF$ DISTRIBUTE R(BLOCK), S(BLOCK), IX(BLOCK)

!HPF$ INDEPENDENT (I) ON S(I)
      DO I = 1, 1000
!HPF$       ATOMIC UPDATE
            R(IX(I)) = R(IX(I)) + S(I)
      END DO
```

F.5.17 The GEOMETRY Directive

The `GEOMETRY` directive is simliar to a `typedef` in C, only it is for data mapping. It allows the user to conveniently change the mappings of many arrays at the same time. It is similar in many ways to the `TEMPLATE` directive, but since it is bound to no particular extent it is sometimes easier to apply.

`!HPF$ GEOMETRY` *geom*`(`d_1 `[,` d_2`, ..., ` d_n`])`
`!HPF$ DISTRIBUTE (` *geom* `) [::]` var_1`[,` var_2`, ... ,` var_m`]`

Where d_i indicates one of the allowable distribution formats.

```
!HPF$ GEOMETRY GBB(BLOCK, CYCLIC)
      REAL A(300,300), B(400,400)
!HPF$ DISTRIBUTE (GBB) ::  A, B
!       if GBB changes then both A and B change
```

附属書G The FORTRAN 77 Local Library

The HPF standard now describes an `EXTRINSIC(LANGUAGE='F77',MODEL='LOCAL')` interface, or `EXTRINSIC(F77_LOCAL)` to use the keyword identification (see Section 11.6 for its description), similar in characteristics to the `EXTRINSIC(LANGUAGE='HPF',MODEL='LOCAL')` and `EXTRINSIC(LANGUAGE='FORTRAN',MODEL='LOCAL')` interfaces. This section describes a set of library routines to make it easier to make use of the `F77_LOCAL` interface when passing distributed array data. These library routines can facilitate, for example, a portable blend of global data parallel code with preexisting FORTRAN 77-based code using explicit message passing calls for interprocessor communication. The FORTRAN 77 Local Library interface described in this section was originally developed as part of Thinking Machines TMHPF and is now supported by Sun Microsystems Inc. For suggestions, requests, or corrections concerning this interface, please contact

Sun Microsystems Inc.
High Performance Computing
M/S UCHLO5-104
5 Omni Way
Chelmsford, MA 01824
f77-local-library@sun.com

G.1 Introduction

The basic constraints for the local model (Section 11.1) together with the `F77_LOCAL`-specific argument passing options (Section 11.6) define the nature of the `F77_LOCAL` interface: how control is to be transferred from a global HPF procedure to a set of local procedures described by an `EXTRINSIC(F77_LOCAL)` procedure interface and how data can be passed between these two types of procedures: by reference or by descriptor, and with or without temporary local reordering of data to satisfy FORTRAN 77 provisions for sequential, contiguous storage of array data in Fortran array element order. These alternative methods of argument passing can be obtained by use of the two special-purpose attributes for extrinsic dummy arguments defined for `LANGUAGE='F77'` routines: `LAYOUT('F77_ARRAY')` (the default) vs. `LAYOUT('HPF_ARRAY')`, and `PASS_BY('*')` (the default) vs. `PASS_BY('HPF_HANDLE')`. However, to take advantage of the option allowing one to pass global HPF array "handles" to local FORTRAN 77 procedures and then obtain information locally about how the local

portion of a given parallel array is actually distributed requires special inquiry routines comparable to the HPF Local Library of functions. Since this library is not only described as a module, but uses many features such as array-valued functions and optional arguments not available in FORTRAN 77 code, it is recommended that a modified FORTRAN 77 interface to this library be provided in the manner described below. Furthermore, there is the problem of describing local portions of parallel arrays in the FORTRAN 77 code used in each local routine called from a global HPF one. Since assumed-shape syntax may not be used, explicit shape arrays are required. But it is common for global distribution of arbitrary sized arrays to result in local portions of arrays that do not have constant shapes on all processors, and the actual extents in each processor cannot necessarily be predicted in advance. In order to allow programmers to obtain axis extent information at run time from the HPF global caller, a special HPF-callable subgrid inquiry subroutine is provided. A FORTRAN 77 callable version of the same routine is also described below, for flexibility in programming.

G.2 Summary

- One HPF-callable subgrid inquiry subroutine

  `HPF_SUBGRID_INFO`

- A set of FORTRAN 77-callable inquiry subroutines

  `F77_SUBGRID_INFO`

  `F77_GLOBAL_ALIGNMENT`

  `F77_GLOBAL_DISTRIBUTION`

  `F77_GLOBAL_TEMPLATE`

  `F77_ABSTRACT_TO_PHYSICAL`

  `F77_PHYSICAL_TO_ABSTRACT`

  `F77_LOCAL_TO_GLOBAL`

  `F77_GLOBAL_TO_LOCAL`

  `F77_LOCAL_BLKCNT`

  `F77_LOCAL_LINDEX`

  `F77_LOCAL_UINDEX`

  `F77_GLOBAL_SHAPE`

  `F77_GLOBAL_SIZE`

  `F77_SHAPE`

  `F77_SIZE`

  `F77_MY_PROCESSOR`

G.3 Global HPF Subgrid Inquiry Routine

The F77_LOCAL library interface includes only one global HPF subroutine, HPF_SUBGRID_INFO, whose implementation should be added as an extension to the standard HPF Library module. Its purpose is to provide per-processor information about the local subgrids of distributed arrays. This information is often critical when passing such arrays to local procedures written in FORTRAN 77, where array argument shapes must be stated explicitly in the local procedure (except in the last dimension; there are "assumed size" but no "assumed shape" arrays) , but may be expressed in terms of arguments passed at run time ("adjustable shape arrays"). Thus the subgrid parameters obtained from this subgrid inquiry routine can be passed as arguments to the local routines and used there to describe the extents of the locally visible portions of global HPF arrays, as the example in Section G.5 will demonstrates.

HPF_SUBGRID_INFO (ARRAY, IERR, DIM, LB, UB, STRIDE, LB_EMBED, UB_EMBED, AXIS_MAP)

Description. Gives local information about local subgrid allocation onto each processor of a distributed array; callable from a global HPF routine.

Class. Inquiry subroutine.

Arguments.

ARRAY	is a nonsequential array of any type, size, shape, or mapping. It is an INTENT (IN) argument.
IERR	is a scalar integer of default kind. It is an INTENT (OUT) argument. Its return value is zero upon successful return and nonzero otherwise. Errors result if local subgrids cannot be expressed as array sections of ARRAY. If any of the optional arguments LB_EMBED, UB_EMBED, or AXIS_MAP is present, then a nonzero value is also returned if the compiler does not organize the local data in serial memory by sequence associating a larger "embedding" array (see Section G.3.1 below for more explanation).
DIM (optional)	is a scalar integer of default kind. It is an INTENT (IN) argument. DIM indicates the axis along which return values are desired. If DIM is not present, values are returned for all axes.
LB (optional)	is an INTENT (OUT), default integer array. If this argument is present, and if the value returned in IERR is zero, the values returned in array LB are the lower bounds in

global coordinates of each processor's subgrid, along one (if DIM is present) or each dimension of ARRAY.

UB (optional) — is an INTENT (OUT), default integer array. If this argument is present, and if the value returned in IERR is zero, the values returned in array UB are the upper bounds in global coordinates of each processor's subgrid, along one (if DIM is present) or each dimension of ARRAY.

STRIDE (optional) — is an INTENT (OUT), default integer array. If this argument is present, and if the value returned in IERR is zero, the values returned in array STRIDE are the strides in local memory between elements of each processor's subgrid, along one (if DIM is present) or each dimension of ARRAY.

LB_EMBED (optional) — is an INTENT (OUT), default integer array. If this argument is present, and if the value returned in IERR is zero, the values returned in array LB_EMBED are the lower bounds in global coordinates of the actual global array elements allocated on each processor, possibly a superset of the user-visible subgrid, along one (if DIM is present) or each dimension of ARRAY.

UB_EMBED (optional) — is an INTENT (OUT), default integer array. If this argument is present, and if the value returned in IERR is zero, the values returned in array UB_EMBED are the upper bounds in global coordinates of the actual global array elements allocated on each processor, possibly a superset of the user-visible subgrid, along one (if DIM is present) or each dimension of ARRAY.

AXIS_MAP (optional) — is a rank 2, INTENT (OUT), default integer array. If this argument is present, its shape must be at least [n,r], where n is the number of processors and r is the rank of ARRAY.

If the value returned in IERR is zero, the values returned in AXIS_MAP(i,1:r) represent the numbers of the axes of the subgrid on processor i from fastest varying to slowest varying, and form a permutation of the sequence 1,2,...,r.

For the last six arguments, LB, UB, STRIDE, LB_EMBED, UB_EMBED, and AXIS_MAP, each array has a first axis of extent at least n, where n is the number of processors, and the first n indices of that axis of each array must be distributed (perhaps via an explicit CYCLIC or BLOCK distribution) one index per processor. If a second dimension is needed, it should be a collapsed axis of extent at least equal to the rank of ARRAY.

If HPF_SUBGRID_INFO is called, and the elements of ARRAY that are local to any particular processor are not representable as an array section of the global user array, then a nonzero value is returned for IERR. Otherwise, if any of the optional arguments LB, UB, or STRIDE is present, then the lower bounds, upper bounds, or strides, respectively, that describe the local array sections are returned in terms of one-based, global coordinates.

G.3.1 Subgrid Inquiries Involving Embedding Arrays

In the common case in which the elements of each local subgrid of the global array argument are distributed across processors, with no overlap, and allocated in local memory like a local FORTRAN 77 array, as a contiguous sequence of elements in Fortran array element order, these three last optional arguments would not be required.

However, some implementations may choose less common layouts in local memory, that involve "embedding" these elements in a larger array section of equal rank that *is* sequence-associated in serial memory. For example, alignment of axes of arrays in different orders may result in a permuting embedding of the subgrid. Or axes of subgrids map be padded with ghost cells, either for stencil optimizations or to achieve same-size subgrids on all nodes.

In variations such as these, we may still view the subgrid as being "embedded" in a sequence associated array which may be accessible in F77_LOCAL operations, if the permutation of axes, shape of any embedding array, and offsets into that array can be obtained at runtime. The last three arguments of HPF_SUBGRID_INFO are provided to allow programmers to obtain this information when it is appropriate, with the help of the IERR flag to signal when this is not the case.

In this mapping, local memory has been allocated for a larger array section, with coordinates (LB_EMBED : UB_EMBED : STRIDE). The coordinates of the *actual* computational elements are limited to the subset (LB : UB : STRIDE). The sequence association is generalized to an arbitrary mapping of axes. Here, AXIS_MAP numbers the axes from fastest varying to slowest varying. If LB_EMBED, UB_EMBED, or AXIS_MAP is specified in a call to HPF_SUBGRID_INFO but ARRAY does not satisfy the assumptions of this mapping model, then a nonzero value is returned for IERR.

G.4 Local FORTRAN 77 Inquiry Routines

Here the F77-callable inquiry subroutines are described briefly. These provide essentially the same capability as the combination of the HPF intrinsic array inquiry functions such as SHAPE and SIZE, together with the HPF LOCAL LIBRARY inquiry routines. The subroutine F77_SUBGRID_INFO serves as a local counterpart to the globally callable subroutine HPF_SUBGRID_INFO described above. In all of the following:

- ARRAY is a dummy argument passed in from a global HPF caller using the LAYOUT ('HPF_ARRAY') attribute and declared within the FORTRAN 77 local subroutine as a scalar integer variable. It is an INTENT (IN) argument.

- DIM is a scalar integer of default kind. It is an INTENT (IN) argument. This argument specifies a particular axis of the global array associated with ARRAY or, if DIM = -1, inquiry is for all axes.

- An "inquiry result" is an INTENT (OUT) argument. If DIM = -1, it is a rank-one array of size equal to at least the rank of the global array associated with ARRAY, returning information associated with all axes. If DIM is positive, the "inquiry result" is a scalar, returning information only for the axis indicated by DIM.

- The arguments are defined in the same way as for the corresponding HPF or HPF_LOCAL routines unless otherwise noted. See the description of HPF_SUBGRID_INFO above and Section 11.7.1 for full specifications of the similarly-named HPF_LOCAL_LIBRARY procedures.

F77_SUBGRID_INFO (ARRAY, IERR1, IERR2, DIM, LB, UB, STRIDE, LB_EMBED, UB_EMBED, AXIS_MAP)

Description. This is a FORTRAN 77-callable version of the HPF subroutine HPF_SUBGRID_INFO.

Arguments.

IERR1 is a scalar integer of default kind. It is an INTENT (OUT) argument. Its return value is zero if LB, UB, and STRIDE were determined successfully and nonzero otherwise.

IERR2 is a scalar integer of default kind. It is an INTENT (OUT) argument. Its return value is zero if LB_EMBED and UB_EMBED were determined successfully and nonzero otherwise.

LB, UB, STRIDE, LB_EMBED, UB_EMBED, AXIS_MAP are "inquiry results" of default integer type. They are the lower and upper bounds and strides of the array sections describing the local data (in terms of global indices), the lower and upper bounds of the embedding arrays (again, in terms of global indices), and the axes of the embedding arrays to which the axes of ARRAY are mapped.

F77_GLOBAL_ALIGNMENT (ALIGNEE, LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES)

Description. This is a FORTRAN 77-callable version of the HPF_LOCAL subroutine GLOBAL_ALIGNMENT. All but the first are INTENT (OUT) arguments whose return values are as specified by the corresponding HPF routine.

Arguments.

`ALIGNEE`	is a dummy argument passed in from global HPF. It is an `INTENT (IN)` argument.
`LB, UB, STRIDE, AXIS_MAP`	are integer arrays of rank one. Their size must be at least equal to the rank of the global HPF array associated with `ALIGNEE`.
`IDENTITY_MAP, DYNAMIC`	are scalar logicals.
`NCOPIES`	is a scalar integer of default kind.

F77_GLOBAL_DISTRIBUTION (DISTRIBUTEE, AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` subroutine `GLOBAL_DISTRIBUTION`. All but the first are `INTENT (OUT)` arguments whose return values are as specified by the corresponding HPF routine.

Arguments.

`DISTRIBUTEE`	is a dummy argument passed in from global HPF. It is an `INTENT (IN)` argument.
`AXIS_TYPE`	is a `CHARACTER*9` array of rank one. Its size must be at least equal to the rank of the global HPF array associated with `DISTRIBUTEE`.
`AXIS_INFO`	is a default integer array of rank one. Its size must be at least equal to the rank of the global HPF array associated with `DISTRIBUTEE`.
`PROCESSORS_RANK`	is a scalar of default integer type.
`PROCESSORS_SHAPE`	is an integer array of rank one. Its size must be at least equal to the value returned by `PROCESSORS_RANK`.

F77_GLOBAL_TEMPLATE (ALIGNEE, TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED, DYNAMIC)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` subroutine `GLOBAL_TEMPLATE`. All but the first are `INTENT (OUT)` arguments whose return values are as specified by the corresponding HPF routine.

Arguments.

`ALIGNEE`	is a dummy argument passed in from global HPF. It is an `INTENT (IN)` argument.

TEMPLATE_RANK	is a scalar integer of default kind.
LB, UB, AXIS_INFO	are integer arrays of rank one. Their size must be at least equal to the rank of the align-target to which the global HPF array associated with ALIGNEE is ultimately aligned.
AXIS_TYPE	is a CHARACTER*10 array of rank one. Its size must be at least equal to the rank of the align-target to which the global HPF array associated with ALIGNEE is ultimately aligned.
NUMBER_ALIGNED	is a scalar integer of default kind.
DYNAMIC	is a scalar logical.

F77_ABSTRACT_TO_PHYSICAL(ARRAY, INDEX, PROC)

Description. This is a FORTRAN 77-callable version of the HPF_LOCAL subroutine ABSTRACT_TO_PHYSICAL.

Arguments.

INDEX	is a rank-one, INTENT (IN), integer array.
PROC	is a scalar, INTENT (OUT), integer.

F77_PHYSICAL_TO_ABSTRACT(ARRAY, PROC, INDEX)

Description. This is a FORTRAN 77-callable version of the HPF_LOCAL subroutine PHYSICAL_TO_ABSTRACT.

Arguments.

PROC	is a scalar, INTENT (IN), integer.
INDEX	is a rank-one, INTENT (OUT), integer array.

F77_LOCAL_TO_GLOBAL(ARRAY, L_INDEX, G_INDEX)

Description. This is a FORTRAN 77-callable version of the HPF_LOCAL subroutine LOCAL_TO_GLOBAL.

Arguments.

L_INDEX	is a rank-one, INTENT (IN), integer array.
G_INDEX	is a rank-one, INTENT (OUT), integer array.

F77_GLOBAL_TO_LOCAL(ARRAY, G_INDEX, L_INDEX, LOCAL, NCOPIES, PROCS)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` subroutine `GLOBAL_TO_LOCAL`.

Arguments.

`G_INDEX`	is a rank-one, `INTENT (IN)`, integer array.
`L_INDEX`	is a rank-one, `INTENT (OUT)`, integer array.
`LOCAL`	is a scalar, `INTENT (OUT)`, logical.
`NCOPIES`	is a scalar, `INTENT (OUT)`, integer.
`PROCS`	is a rank-one, integer array whose size is at least the number of processors that hold copies of the identified element.

F77_LOCAL_BLKCNT(L_BLKCNT, ARRAY, DIM, PROC)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` function `LOCAL_BLKCNT`.

Arguments.

`L_BLKCNT`	is an "inquiry result" of type integer.
`PROC`	is a scalar integer of default kind. It must be a valid processor number or, if `PROC = -1`, the value returned by `F77_MY_PROCESSOR()` is implied.

F77_LOCAL_LINDEX(L_LINDEX, ARRAY, DIM, PROC)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` function `LOCAL_LINDEX`.

Arguments.

`L_LINDEX`	is a rank-one, integer array of size equal to at least the value returned by `F77_LOCAL_BLKCNT`.
`DIM`	may not be -1.
`PROC`	is a scalar integer of default kind. It must be a valid processor number or, if `PROC = -1`, the value returned by `F77_MY_PROCESSOR()` is implied.

F77_LOCAL_UINDEX(L_UINDEX, ARRAY, DIM, PROC)

Description. This is a FORTRAN 77-callable version of the HPF_LOCAL function LOCAL_UINDEX.

Arguments.

L_UINDEX	is a rank-one, integer array of size equal to at least the value returned by F77_LOCAL_BLKCNT.
DIM	may not be -1.
PROC	is a scalar integer of default kind. It must be a valid processor number or, if PROC = -1, the value returned by F77_MY_PROCESSOR() is implied.

F77_GLOBAL_SHAPE(SHAPE, ARRAY)

Description. This is a FORTRAN 77-callable version of the HPF_LOCAL function GLOBAL_SHAPE.

Arguments.

SHAPE	is a rank-one, integer array of size equal to at least the rank of the global array associated with ARRAY. Its return value is the shape of that global array.

F77_GLOBAL_SIZE(SIZE, ARRAY, DIM)

Description. This is a FORTRAN 77-callable version of the HPF_LOCAL function GLOBAL_SIZE.

Arguments.

SIZE	is a scalar integer equal to the extent of axis DIM of the global array associated with ARRAY or, if DIM = -1, the total number of elements in that global array.

F77_SHAPE(SHAPE, ARRAY)

Description. This is a FORTRAN 77-callable version of the HPF intrinsic SHAPE, as it would behave as called from HPF_LOCAL.

Arguments.

SHAPE	is a rank-one, integer array of size equal to at least the rank of the subgrid associated with ARRAY. Its return value is the shape of that subgrid.

F77_SIZE(SIZE, ARRAY, DIM)

Description. This is a FORTRAN 77-callable version of the HPF intrinsic `SIZE`, as it would behave as called from `HPF_LOCAL`.

Arguments.

`SIZE`	is a scalar integer equal to the extent of axis `DIM` of the subgrid associated with `ARRAY` or, if `DIM = -1`, the total number of elements in that subgrid.

F77_MY_PROCESSOR(MY_PROC)

Description. This is a FORTRAN 77-callable version of the `HPF_LOCAL` function `MY_PROCESSOR`.

Arguments.

`MY_PROC`	is a scalar, `INTENT (OUT)`, integer. Its value is the identifying number of the physical processor from which this call is made.

G.5 Programming Example Using HPF_SUBGRID_INFO

G.5.1 HPF Caller

```
      PROGRAM EXAMPLE
! Declare the data array and a verification copy
      INTEGER, PARAMETER :: NX = 100, NY = 100
      REAL, DIMENSION(NX,NY) :: X, Y
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X, Y
! The global sum will be computed
! by forming partial sums on the processors
      REAL PARTIAL_SUM(NUMBER_OF_PROCESSORS())
!HPF$ DISTRIBUTE PARTIAL_SUM(BLOCK)
! Local subgrid parameters are declared per processor
! for a rank-two array
      INTEGER, DIMENSION(NUMBER_OF_PROCESSORS(),2) ::
     & LB, UB, NUMBER
!HPF$ DISTRIBUTE(BLOCK,*) :: LB, UB, NUMBER
! Define interfaces
      INTERFACE
        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL1
     &   ( LB1, UB1, LB2, UB2, NX, X )
! Arrays LB1, UB1, LB2, UB2, and X are passed by default
```

```
! as LAYOUT('F77_ARRAY') and PASS_BY('*')
        INTEGER, DIMENSION(:) :: LB1, UB1, LB2, UB2
        INTEGER NX
        REAL X(:,:)
!HPF$ DISTRIBUTE(BLOCK) :: LB1, UB1, LB2, UB2
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X
        END
        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL2(N,X,R)
! Arrays N, X, and R are passed by default
! as LAYOUT('F77_ARRAY') and PASS_BY('*')
        INTEGER N(:)
        REAL X(:,:), R(:)
!HPF$ DISTRIBUTE N(BLOCK)
!HPF$ DISTRIBUTE X(BLOCK,BLOCK)
!HPF$ DISTRIBUTE R(BLOCK)
        END
      END INTERFACE

! Determine result using only global HPF
      ! Initialize values
      FORALL (I=1:NX,J=1:NY) X(I,J) = I + (J-1) * NX
      ! Determine and report global sum
      PRINT *, 'GLOBAL HPF RESULT: ',SUM(X)
! Determine result using local subroutines
      ! Initialize values ( assume stride = 1 )
      CALL HPF_SUBGRID_INFO( Y, IERR, LB=LB, UB=UB )
      IF (IERR.NE.0) STOP 'ERROR!'
      CALL LOCAL1( LB(:,1), UB(:,1), LB(:,2), UB(:,2), NX, Y )
      ! Determine and report global sum
      NUMBER = UB - LB + 1
      CALL LOCAL2 ( NUMBER(:,1) * NUMBER(:,2) , Y , PARTIAL_SUM )
      PRINT *, 'F77_LOCAL RESULT #1 : ',SUM(PARTIAL_SUM)
      END
```

G.5.2 FORTRAN 77 Callee

```
      SUBROUTINE LOCAL1( LB1, UB1, LB2, UB2, NX, X )
! The global actual arguments passed to LB1, UB1, LB2, and UB2
! have only one element apiece and so can be treated as scalars
! in the local Fortran 77 procedures
      INTEGER LB1, UB1, LB2, UB2
! NX contains the global extent of the first dimension
```

```
! of the global array associated with local array X
      INTEGER NX
! Note that X may have no local elements.
      REAL X ( LB1 : UB1 , LB2 : UB2 )
! Initialize the elements of the array, if any
      DO J = LB2, UB2
        DO I = LB2, UB2
          X(I,J) = I + (J-1) * NX
        END DO
      END DO
      END

      SUBROUTINE LOCAL2(N,X,R)
! Here, the rank of the original array is unimportant
! Only the total number of local elements is needed
!     INTEGER N
      REAL X(N), R
! If N is zero, local array X has no elements, but R
! still computes the correct local sum
      R = 0.
      DO I = 1, N
        R = R + X(I)
      END DO
      END
```

G.6 Programming Example Using F77-Callable Inquiry Subroutines

This example performs only the initialization of the above example. It illustrates use of the F77-callable inquiry routines on descriptors passed from HPF, as well as the addressing of uncompressed local subgrid data in terms of "embedding arrays."

G.6.1 HPF Caller

```
      PROGRAM EXAMPLE
      INTEGER, PARAMETER :: NX = 100, NY = 100
      REAL, DIMENSION(NX,NY) :: X
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X
! Local subgrid parameters are declared per processor
! for a rank-two array
      INTEGER, DIMENSION(NUMBER_OF_PROCESSORS(),2) ::
     & LB, UB, LB_EMBED, UB_EMBED
!HPF$ DISTRIBUTE(BLOCK,*) :: LB, UB, LB_EMBED, UB_EMBED
! Define interfaces
```

```
      INTERFACE
        EXTRINSIC(F77_LOCAL) SUBROUTINE LOCAL1(
     &   LB1, UB1, LB_EMBED1, UB_EMBED1,
     &   LB2, UB2, LB_EMBED2, UB_EMBED2, X, X_DESC )
        INTEGER, DIMENSION(:) ::
     &   LB1, UB1, LB_EMBED1, UB_EMBED1,
     &   LB2, UB2, LB_EMBED2, UB_EMBED2
! X is passed twice, both times without local reordering.
! First, it is passed by reference for accessing array elements.
        REAL, DIMENSION(:,:), LAYOUT('HPF_ARRAY'),
     &          PASS_BY('*')                    :: X
! It is also passed by descriptor for use in F77 LOCAL
! LIBRARY subroutines only.
        REAL, DIMENSION(:,:), LAYOUT('HPF_ARRAY'),
     &          PASS_BY('HPF_HANDLE')           :: X_DESC
!HPF$ DISTRIBUTE(BLOCK) :: LB1, UB1, LB_EMBED1, UB_EMBED1
!HPF$ DISTRIBUTE(BLOCK) :: LB2, UB2, LB_EMBED2, UB_EMBED2
!HPF$ DISTRIBUTE(BLOCK,BLOCK) :: X
        END
      END INTERFACE
! Initialize values
! ( Assume stride = 1 and no axis permutation )
      CALL HPF_SUBGRID_INFO( X, IERR,
     & LB=LB, LB_EMBED=LB_EMBED,
     & UB=UB, UB_EMBED=UB_EMBED)
      IF (IERR.NE.0) STOP 'ERROR!'
      CALL LOCAL1(
     & LB(:,1), UB(:,1), LB_EMBED(:,1), UB_EMBED(:,1),
     & LB(:,2), UB(:,2), LB_EMBED(:,2), UB_EMBED(:,2), X, X )
      END
```

G.6.2 FORTRAN 77 Callee

```
      SUBROUTINE LOCAL1(
     & LB1, UB1, LB_EMBED1, UB_EMBED1,
     & LB2, UB2, LB_EMBED2, UB_EMBED2, X, X_DESC )
      INTEGER LB1, UB1, LB_EMBED1, UB_EMBED1
      INTEGER LB2, UB2, LB_EMBED2, UB_EMBED2
! The subgrid has been passed in its 'embedded' form
      REAL X ( LB_EMBED1 : UB_EMBED1 , LB_EMBED2 : UB_EMBED2 )
! Locally X_DESC is declared as an INTEGER
      INTEGER X_DESC
```

```
! Get the global extent of the first axis
! This is an HPF_LOCAL type of inquiry routine with an
! 'F77_' prefix
      CALL F77_GLOBAL_SIZE(NX,X,1)
! Otherwise, initialize elements of the array
! Loop only over actual array elements
      DO J = LB2, UB2
        DO I = LB2, UB2
          X(I,J) = I + (J-1) * NX
        END DO
      END DO
      END
```