High Performance Fortran Language Specification

High Performance Fortran Forum

November 10, 1994
Version 1.1

The High Performance Fortran Forum (HPFF), with participation from over 40 organizations, met from March 1992 to March 1993 to discuss and 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.

This is the Final Report, Version 1.1, of the High Performance Fortran Forum 1994 meetings. This document contains all the technical features proposed for the version of the language known as HPF 1.1 This copy of the draft was processed by LaTeX on November 11, 1994.

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.

Contents

Acknowledgments

Since its introduction over three decades ago, Fortran has been the language of choice for scientific programming for sequential computers. Exploiting the full capability of modern architectures, however, increasingly requires more information than ordinary FORTRAN 77 or Fortran 90 programs provide. This information applies to such areas as:

- Opportunities for parallel execution;
- Type of available parallelism — MIMD, SIMD, or some combination;
- Allocation of data among individual processor memories; and
- Placement of data within a single processor.

The High Performance Fortran Forum (HPFF) was founded as a coalition of industrial and academic groups working to suggest a set of standard extensions to Fortran to provide the necessary information. Its intent was to develop 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 included most vendors delivering parallel machines, a number of government laboratories, and many university research groups. Public input was encouraged to the greatest extent possible. The result of this project is this document, intended to be a language specification portable from workstations to massively parallel supercomputers while being able to express the algorithms needed to achieve high performance on specific architectures.

0.1 HPFF Acknowledgements

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).

0.2 HPFF94 Acknowledgements

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	Miss. State University / 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.

Section 1

Overview

This document specifies the form and establishes the interpretation of programs expressed in the High Performance Fortran (HPF) language. It is designed as a set of extensions and modifications to the established International Standard for Fortran (ISO/IEC 1539:1991(E) and ANSI X3.198-1992), informally referred to as "Fortran 90" ([12]). Many sections of this document reference related sections of the Fortran 90 standard to facilitate its incorporation into new standards, should ISO and national standards committees deem that desirable.

1.1 Goals and Scope of High Performance Fortran

The goals of HPF, as defined at an early HPFF meeting, were to define language extensions and feature selection for Fortran supporting:

- Data parallel programming (defined as single threaded, global name space, and loosely synchronous parallel computation);
- Top performance on MIMD and SIMD computers with non-uniform memory access costs (while not impeding performance on other machines); and
- Code tuning for various architectures.

The `FORALL` construct and several new intrinsic functions were designed primarily to meet the first goal, while the data distribution features and some other directives are targeted toward the second goal. Extrinsic procedures allow access to low-level programming in support of the third goal, although performance tuning using the other features is also possible.

A number of subsidiary goals were also established:

- Deviate minimally from other standards, particularly those for FORTRAN 77 and Fortran 90;
- Keep the resulting language simple;
- Define open interfaces to other languages and programming styles;
- Provide input to future standards activities for Fortran and C;
- Encourage input from the high performance computing community through widely distributed language drafts;

- Produce validation criteria;
- Present the final proposals in November 1992 and accept the final draft in January 1993;
- Make compiler availability feasible in the near term with demonstrated performance on an HPF test suite; and
- Leave an evolutionary path for research.

These goals were quite aggressive when they were adopted in March 1992, and led to a number of compromises in the final language. In particular, support for explicit MIMD computation, message-passing, and synchronization was limited due to the difficulty in forming a consensus among the participants. We hope that future efforts will address these important issues.

1.2 Fortran 90 Binding

HPF is an extension of Fortran 90. The array calculation and dynamic storage allocation features of Fortran 90 make it a natural base for HPF. The new HPF language features fall into four categories with respect to Fortran 90:

- New directives;
- New language syntax;
- Library routines; and
- Language changes and restrictions.

The new directives are structured comments that suggest implementation strategies or assert facts about a program to the compiler. They may affect the efficiency of the computation performed, but do not change the value computed by the program. The form of the HPF directives has been chosen so that a future Fortran standard may choose to include these features as full statements in the language by deleting the initial comment header.

A few new language features, including the `FORALL` statement and a few intrinsic functions, are also defined. They were made first-class language constructs rather than comments because they can affect the interpretation of a program, for example by returning a value used in an expression. These are proposed as direct extensions to the Fortran 90 syntax and interpretation.

The HPF library of computational functions defines a standard interface to routines that have proven valuable for high performance computing including additional reduction functions, combining scatter functions, prefix and suffix functions, and sorting functions.

Two small changes are made in the Fortran 90 specification. First, a `DIM` argument is added to the `MINLOC` and `MAXLOC` routines. Second, in the list of keyword specifiers for the I/O `INQUIRE` statement, the types of `RECL`, `NEXTREC`, and `IOLENGTH` are changed to *scalar-integer-variable* (from *scalar-default-integer-variable*) in order to allow for very long files that may occur in large parallel applications.

Full support of Fortran sequence and storage association is not compatible with the data distribution features of HPF. Some restrictions on the use of sequence and storage association are defined. These restrictions may in turn require insertion of HPF directives into standard Fortran 90 programs in order to preserve correct semantics.

1.3 New Features in High Performance Fortran

HPF extends Fortran 90 in several areas, including:

- Data distribution features;
- Data parallel execution features;
- Extended intrinsic functions and standard library;
- `EXTRINSIC` procedures;
- Changes in sequence and storage association.

In addition, a subset of HPF suitable for earlier implementation is defined. The following subsections give short overviews of these areas.

In addition to the features that became part of HPF, the HPFF committee considered and rejected many proposals. Suggestions that the committee considered particularly promising for future language efforts to pursue have been collected in a companion document, the HPF Journal of Development [15]. Section 1.7 below gives an overview of this document.

1.3.1 Data Distribution Features

Modern parallel and sequential architectures attain their highest speed when the data accessed exhibits locality of reference. The sequential storage order implied by FORTRAN 77 and Fortran 90 often conflicts with the locality demanded by the architecture. To avoid this, HPF includes features which describe the collocation of data (`ALIGN`) and the partitioning of data among memory regions or abstract processors (`DISTRIBUTE`). Compilers may interpret these annotations to improve storage allocation for data, subject to the constraint that semantically every data object has a single value at any point in the program. In all cases, users should expect the compiler to arrange the computation to minimize communication while retaining parallelism. Section 3 describes the distribution features.

1.3.2 Data Parallel Execution Features

To express parallel computation explicitly, HPF offers a new statement and a new directive. The `FORALL` construct expresses assignments to sections of arrays; it is similar in many ways to the array assignment of Fortran 90, but allows more general sections and computations to be specified. The `INDEPENDENT` directive asserts that the statements in a particular section of code do not exhibit any sequentializing dependences; when properly used, it does not change the semantics of the construct, but may provide more information to the language processor to allow optimizations. Section 4 describes these features.

1.3.3 Extended Intrinsic Functions and Standard Library

Experience with massively parallel machines has identified several basic operations that are very valuable in parallel algorithm design. The Fortran 90 array intrinsics anticipated some of these, but not all. HPF adds several classes of parallel operations to the language definition as intrinsic functions and as standard library functions. In addition, several system inquiry functions useful for controlling parallel execution are provided in HPF. Section 5 describes these functions and subroutines.

1.3.4 Extrinsic Procedures

Because HPF is designed as a high-level, machine-independent language, there are certain operations that are difficult or impossible to express directly. For example, many applications benefit from finely-tuned systolic communications on certain machines; HPF's global address space does not express this well. Extrinsic procedures define an explicit interface to procedures written in other paradigms, such as explicit message-passing subroutine libraries. Section 6 describes this interface. Annex A gives a specific interface for HPF_LOCAL routines, for HPF_SERIAL routines, and for Fortran 90.

1.3.5 Sequence and Storage Association

A goal of HPF was to maintain compatibility with Fortran 90. Full support of Fortran sequence and storage association, however, is not compatible with the goal of high performance through distribution of data in HPF. Some forms of associating subprogram dummy arguments with actual values make assumptions about the sequence of values in physical memory which may be incompatible with data distribution. Certain forms of `EQUIVALENCE` statements are recognized as requiring a modified storage association paradigm. In both cases, HPF provides a directive to assert that full sequence and storage association for affected variables must be maintained. In the absence of such explicit directives, reliance on the properties of association is not allowed. An optimizing compiler may then choose to distribute any variables across processor memories in order to improve performance. To protect program correctness, a given implementation should provide a mechanism to ensure that all such default optimization decisions are consistent across an entire program. Section 7 describes the restrictions and directives related to storage and sequence association.

1.4 Fortran 90 and Subset HPF

An important goal for HPF is early compiler availability. Because full Fortran 90 compilers may not be available in a timely fashion on all platforms and implementation of some HPF features is more complex than others, we have defined Subset HPF. Users who are most concerned about multi-machine portability may choose to stay within this subset initially. This subset language includes the Fortran 90 array language, dynamic storage allocation, and long names as well as the MIL-STD-1753 features ([29]), which are already commonly used with FORTRAN 77 programs. The subset does not include features of Fortran 90, such as generic functions and free source form, that are not closely related to high performance on parallel machines. Section 8 describes Subset HPF.

1.5 Notation

This document uses the same notation as the Fortran 90 standard. In particular, the same conventions are used for syntax rules. BNF descriptions of language features are given in the style used in the Fortran 90 standard. To distinguish HPF syntax rules from Fortran 90 rules, each HPF rule has an identifying number of the form H*snn*, where *s* is a one-digit major section number and *nn* is a one- or two-digit sequence number. The syntax rules are also collected in Annex C. Nonterminals not defined in this document are defined in the Fortran 90 standard. Also note that certain technical terms such as "storage unit" are defined by the Fortran 90 standard; Annex D identifies the Fortran 90 rules defining these nonterminals. References in parentheses in the text refer to the Fortran 90 standard.

Rationale. Throughout this document, material explaining the rationale for including features, choosing particular feature definitions, and other decisions is set off in this format. Readers interested in the language definition only may wish to skip these sections, while readers interested in language design may want to read them more carefully. (*End of rationale.*)

Advice to users. Throughout this document, material that is primarily commentary for users (including most examples of syntax and interpretation) is set off in this format. Readers interested in technical material only may wish to skip these sections, while readers wanting a more basic approach may want to read them more carefully. (*End of advice to users.*)

Advice to implementors. Throughout this document, material that is primarily commentary for implementors is set off in this format. Readers interested in the language definition only may wish to skip these sections, while readers interested in compiler implementation may want to read them more carefully. (*End of advice to implementors.*)

1.6 HPF-Conforming and Subset-Conforming

An executable program is HPF-conforming if it uses only those forms and relationships described in this document and if the program has an interpretation according to this document. A program unit is HPF-conforming if it can be included in an executable program in a manner that allows the executable program to be HPF-conforming.

An executable program is Subset-conforming if it uses only the forms and relationships described in this document for Subset HPF (Section 8) and if it has an interpretation under the constraints of Subset HPF. A program unit is Subset-conforming if it can be included in an executable program in a manner that allows the executable program to be Subset-conforming.

(The above definitions were adapted from the Fortran 90 standard.)

1.7 Journal of Development

The HPFF committee considered many proposals, and rejected some that had merit due to external factors (such as lack of agreement in committee). The most promising of these features were collected in the HPF Journal of Development [15]. This section summarizes some of the more detailed proposals.

1.7.1 VIEW Directive

One proposal suggested a directive for relating processor arrangements to each other. This ability is extremely useful in certain applications which use interacting one- and two-dimensional arrays, and has applications for problems consisting of several disjoint data-parallel parts. This feature was carefully discussed, and the committee felt that it was important; however, questions of its implementation complexity eventually caused its rejection.

1.7.2 Nested WHERE Statements

One proposal suggested allowing `WHERE` statements and constructs to be nested within each other. The committee felt that the feature was useful, but declined to include it in HPF because they felt it was too large a change to make to the base language.

1.7.3 EXECUTE-ON-HOME and LOCAL-ACCESS Directives

One proposal suggested a method for specifying the processor(s) to execute a given statement. The same proposal suggested a method for identifying data references which would be mapped to the same processor. In essence, both methods added new directives similar to `INDEPENDENT` (see Section 4.4). Like `INDEPENDENT`, these directives provided information that a compiler might find useful in optimizing the program. Although the committee felt this was an important area to investigate, the proposals were rejected due to technical flaws.

1.7.4 Elemental Reference of Pure Procedures

One proposal suggested allowing elemental invocation of pure procedures (see Section 4.3) under certain conditions. The essential idea was that functions with scalar arguments which could be guaranteed to have no side effects could be invoked elementally, as are intrinsic functions such as `SIN`. The proposal was rejected in a narrow vote, in part because it was seen as too large a change to Fortran 90. After its rejection, the committee voted unanimously to recommend that the ANSI X3J3 committee consider user-defined elemental functions for a future version of Fortran.

1.7.5 Parallel I/O

HPF is primarily designed to obtain high performance on massively parallel computers. Such massively parallel machines also need massively parallel input and output. Accordingly, there were three major proposals to include explicitly parallel I/O features in HPF, as well as several minor variations on the same theme. After much debate, HPFF voted *not* to include I/O extensions in the first version of HPF. (NOTE, however, that HPF1.1 defines changes to Fortran 90 data type of a few of the I/O keyword inquiry specifiers to allow for the possibility of very large files. See section 1.2 on Fortran 90 Binding earlier in this chapter.)

The arguments for not making further extensions or changes for parallel I/O in HPF included:

- The diversity of current parallel I/O systems does not suggest any portable abstraction of I/O useful in a language model.
- Fortran I/O is already highly expressive.
- The HPF compiler can optimize the I/O when writing distributed arrays without any extensions to the source language.
- The management of distributed files (and their implementation) is a matter for the operating system, not the language.

Moreover the current lack of extensions does *not* limit features that may be added by system vendors. In particular:

- Vendors are allowed to implement any I/O extensions to the language they may wish. Indeed this would be impossible to prevent. There are simply no special I/O mechanisms mandated by HPF.
- The HPF run-time system may use whatever facilities the operating system provides for accessing "high performance" files, though the HPF language contains no I/O extensions that specifically describe such access.

1.8 HPF2 Scope of Activities Document

As part of the HPFF94 activities, an additional document, entitled "HPF2: Scope of Activities", was created, with the intent of defining the set of potential added features to be considered in a new HPF development project. The document includes a variety of benchmark Fortran codes that seem to require features not currently present in HPF in order to achieve high performance on distributed memory parallel machines. The document also includes a discussion of potential language extensions that could be added to HPF to facilitate expression of these algorithms. In addition, the notion of creating a kernel subset of HPF is introduced.

1.9 Organization of this Document

Section 1, this section, presents an overview of HPF.

Section 2 sets out some basics of HPF, including:

- The reasons for using Fortran 90 as a base language;
- A partial cost model for HPF programs; and
- Lexical rules for HPF directives.

Section 3 describes the facilities for data partitioning in HPF. These include:

- The distribution model;
- Features for distributing array elements among processors;
- Features for aligning array elements which are accessed together; and
- Features for mapping `ALLOCATABLE` arrays, pointers, and dummy procedure arguments.

Section 4 describes the explicitly parallel statement types in HPF. These include:

- The single- and multi-statement forms of the `FORALL` parallel construct;
- Pure functions callable from within `FORALL`; and
- The `INDEPENDENT` assertion for loops.

Section 5 describes new standard functions available in HPF. These include:

- Inquiry intrinsic functions to check system and data partitioning status;

- New computational intrinsic functions and extensions to existing intrinsic functions; and
- A standard library of computational and inquiry functions.

Section 6 describes extrinsic procedures in HPF, particularly the `EXTRINSIC` procedure interface. The material in Annex A builds on this interface.

Section 7 describes the treatment of sequence and storage association in HPF. This includes:

- Limitations on storage association of explicitly distributed variables; and
- Limitations on sequence association of explicitly distributed variables.

Section 8 describes Subset HPF, which may be implemented more quickly than full HPF. This includes:

- A list of Fortran 90 features that are in Subset HPF;
- A list of HPF features that are *not* in Subset HPF; and
- Discussions of why these decisions were made.

Annex A describes a binding for a local execution model for use as an `EXTRINSIC` option. The model implements the Single Program Multiple Data programming paradigm, which has wide (but not universal) applicability.

Annex C collects the grammar and syntactic constraints for HPF defined in the main text of this document.

Annex D cross-references the BNF terminals and nonterminals defined and used in this document.

The Bibliography provides references to various HPF sources:

- Fortran standards;
- Fortran implementations;
- Books about Fortran 90; and
- Technical papers.

Section 2

High Performance Fortran Terms and Concepts

This Section presents some rationale for the selection of Fortran 90 as HPF's base language, HPF's model of computation, and the high level syntax and lexical rules for HPF directives.

2.1 Fortran 90

The facilities for array computation in Fortran 90 make it particularly suitable for programming scientific and engineering numerical calculations on high performance computers. Indeed, some of these facilities are already supported in compilers from a number of vendors. The introductory overview in the Fortran 90 standard states:

> *Operations for processing whole arrays and subarrays (array sections) are included in the language for two principal reasons: (1) these features provide a more concise and higher level language that will allow programmers more quickly and reliably to develop and maintain scientific/engineering applications, and (2) these features can significantly facilitate optimization of array operations on many computer architectures.*
>
> — Fortran Standard (page xiii)

Other features of Fortran 90 that improve upon the features provided in FORTRAN 77 include:

- Additional storage classes of objects. The new storage classes such as allocatable, automatic, and assumed-shape objects as well as the pointer facility of Fortran 90 add significantly to those of FORTRAN 77 and should reduce the use of FORTRAN 77 constructs that can lead to less than full computational speed on high performance computers, such as `EQUIVALENCE` between array objects, `COMMON` definitions with non-identical array definitions across subprograms, and array reshaping transformations between actual and dummy arguments.

- Support for a modular programming style. The module facilities of Fortran 90 enable the use of data abstractions in software design. These facilities support the specification of modules, including user-defined data types and structures, defined operators on those types, and generic procedures for implementing common algorithms to be used on a variety of data structures. In addition to modules, the definition of interface blocks enables the application programmer to specify subprogram interfaces explicitly, allowing a high quality compiler to use the information specified to provide better checking and optimization at the interface to other subprograms.

- Additional intrinsic procedures. Fortran 90 includes the definition of a large number of new intrinsic procedures. Many of these support mathematical operations on arrays, including the construction and transformation of arrays. Also, there are numerical accuracy intrinsic procedures designed to support numerical programming, and bit manipulation intrinsic procedures derived from MIL-STD-1753.

HPF conforms to Fortran 90 except for additional restrictions placed on the use of storage and sequence association. Because of the effort involved in producing a full Fortran 90 compiler, HPF is defined at two levels: Subset HPF and full HPF. Subset HPF is a subset of Fortran 90 with a subset of the HPF extensions. HPF is Fortran 90 (with the restrictions noted in Section 7) with all of the HPF language features.

2.2 The HPF Model

An important goal of HPF is to achieve code portability across a variety of parallel machines. This requires not only that HPF programs compile on all target machines, but also that a highly-efficient HPF program on one parallel machine be able to achieve reasonably high efficiency on another parallel machine with a comparable number of processors. Otherwise, the effort spent by a programmer to achieve high performance on one machine would be wasted when the HPF code is ported to another machine. Although SIMD processor arrays, MIMD shared-memory machines, and MIMD distributed-memory machines use very different low-level primitives, there is broad similarity with respect to the fundamental factors that affect the performance of parallel programs on these machines. Thus, achieving high efficiency across different parallel machines with the same high level HPF program is a feasible goal. While describing a full execution model is beyond the scope of this language specification, we focus here on two fundamental factors and show how HPF relates to them:

- The parallelism inherent in a computation; and
- The communication inherent in a computation.

The quantitative cost associated with each of these factors is machine dependent; vendors are strongly encouraged to publish estimates of these costs in their system documentation. Note that, like any execution model, these may not reflect all of the factors relevant to performance on a particular architecture.

The parallelism in a computation can be expressed in HPF by the following constructs:

- Fortran 90 array expressions and assignment (including masked assignment in the `WHERE` statement);
- Array intrinsics, including both the Fortran 90 intrinsics and the new intrinsic functions;
- The `FORALL` statement; and
- The `INDEPENDENT` assertion on `DO` loops.

These features allow a user to specify explicitly potential data parallelism in a machine-independent fashion. The purpose of this section is to clarify some of the performance implications of these features, particularly when they are combined with the HPF data distribution features. In addition, EXTRINSIC procedures provide an escape mechanism in HPF to allow the use of efficient machine-specific primitives by using another programming paradigm. Because the resulting model of computation is inherently outside the realm of data-parallel programming, we will not discuss this feature further in this section.

A compiler may choose not to exploit information about parallelism, for example because of lack of resources or excessive overhead. In addition, some compilers may detect parallelism in sequential code by use of dependence analysis. This document does not discuss such techniques.

The interprocessor or inter-memory data communication that occurs during the execution of an HPF program is partially determined by the HPF data distribution directives in Section 3. The compiler will determine the actual mapping of data objects to the physical machine and will be guided in this by the directives. The actual mapping and the computation specified by the program determine the needed actual communication, and the compiler will generate the code required to perform it. In general, if two data references in an expression or assignment are mapped to different processors or memory regions then communication is required to bring them together. The following examples illustrate how this may occur.

Clearly, there is a tradeoff between parallelism and communication. If all the data are mapped to one processor's local memory, then a sequential computation with no communication is possible, although the memory of one processor may not suffice to store all the program's data. Alternatively, mapping data to multiple processors' local memories may permit computational parallelism but also may introduce communications overhead. The optimal resolution of such conflicts is very dependent on the architecture and underlying system software.

The following examples illustrate simple cases of communication, parallelism, and their interaction. Note that the examples are chosen for illustration and do not necessarily reflect efficient data layouts or computational methods for the program fragments shown. Rather, the intent is to derive lower bounds on the amount of communication that are needed to implement the given computations as they are written. This gives some indication of the maximum possible efficiency of the computations on any parallel machine. A particular system may not achieve this efficiency due to analysis limitations, or may disregard these bounds if other factors determine the performance of the code.

2.2.1 Simple Communication Examples

The following examples illustrate the communication requirements of scalar assignment statements. The purpose is to illustrate the implications of data distribution specifications on communication requirements for parallel execution. The explanations given do not necessarily reflect the actual compilation process.

Consider the following statements:

```
      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)                   ! Assignment 1
      x(i) = y(i+1)                 ! Assignment 2
      a(i) = c(i)                   ! Assignment 3
      a(i) = a(i-1) + a(i) + a(i+1) ! Assignment 4
      c(i) = c(i-1) + c(i) + c(i+1) ! Assignment 5
      x(i) = y(i)                   ! Assignment 6
      a(i) = a(inx(i)) + b(inx(i))  ! Assignment 7
```

In this example, the `PROCESSORS` directive specifies a linear arrangement of 10 processors. The `DISTRIBUTE` directives recommend to the compiler that the arrays `a`, `b`, and `inx` should be distributed among the 10 processors with blocks of 100 contiguous elements per processor. The array `c` is to be cyclically distributed among the processors with `c(1)`, `c(11)`, ..., `c(991)` mapped onto processor `procs(1)`; `c(2)`, `c(12)`, ..., `c(992)` mapped onto processor `procs(2)`; and so on. The complete mapping of arrays `x` and `y` onto the processors is not specified, but their relative alignment is indicated by the `ALIGN` directive. The `ALIGN` statement causes `x(i)` and `y(i+1)` to be stored on the same processor for all values of `i`, regardless of the actual distribution chosen by the compiler for `x` and `y` (`y(0)` and `y(1)` are not aligned with any element of `x`). The `PROCESSORS`, `DISTRIBUTE`, and `ALIGN` directives are discussed in detail in Section 3.

In Assignment 1 (`a(i) = b(i)`), the identical distribution of `a` and `b` ensures that for all `i`, `a(i)` and `b(i)` are mapped to the same processor. Therefore, the statement requires no communication.

In Assignment 2 (`x(i) = y(i+1)`), there is no inherent communication. In this case, the relative alignment of the two arrays matches the assignment statement for any actual distribution of the arrays.

Although Assignment 3 (`a(i) = c(i)`) looks very similar to the first assignment, the communication requirements are very different due to the different distributions of `a` and `c`. Array elements `a(i)` and `c(i)` are mapped to the same processor for only 10% of the possible values of `i`. (This can be seen by inspecting the definitions of `BLOCK` and `CYCLIC` in Section 3.) The elements are located on the same processor if and only if $\lfloor (i-1)/100 \rfloor = (i-1) \bmod 10$. For example, the assignment involves no inherent communication (i.e., both `a(i)` and `c(i)` are on the same processor) if $i = 1$ or $i = 102$, but does require communication if $i = 2$.

In Assignment 4 (`a(i) = a(i-1) + a(i) + a(i+1)`), the references to array `a` are all on the same processor for about 98% of the possible values of `i`. The exceptions to this are $i = 100k$ for any $k = 1, 2, \ldots, 9$, (when `a(i)` and `a(i-1)` are on `procs(k)` and `a(i+1)` is on `procs(k+1)`) and $i = 100k + 1$ for any $k = 1, 2, \ldots, 9$ (when `a(i)` and `a(i+1)` are on `procs(k+1)` and `a(i-1)` is on `procs(k)`). Thus, except for "boundary" elements on each processor, this statement requires no inherent communication.

Assignment 5, `c(i) = c(i-1) + c(i) + c(i+1)`, while superficially similar to Assignment 4, has very different communication behavior. Because the distribution of `c` is `CYCLIC` rather than `BLOCK`, the three references `c(i)`, `c(i-1)`, and `c(i+1)` are mapped to three distinct processors for any value of `i`. Therefore, this statement requires communication for at least two of the right-hand side references, regardless of the implementation strategy.

The final two assignments have very limited information regarding the communication requirements. In Assignment 6 (`x(i) = y(i)`) the only information available is that `x(i)` and y(i+1) are on the same processor; this has no logical consequences for the relationship between x(i) and y(i). Thus, nothing can be said regarding communication in the statement without further information. In Assignment 7 (a(i) = a(inx(i)) + b(inx(i))), it can be proved that a(inx(i)) and b(inx(i)) are always mapped to the same processor. Similarly, it is easy to deduce that a(i) and inx(i) are mapped together. Without knowledge of the values stored in inx, however, the relation between a(i) and a(inx(i)) is unknown, as is the relationship between a(i) and b(inx(i)).

The inherent communication for a sequence of assignment statements is the union of the communication requirements for the individual statements. An array element used in several statements contributes to the total inherent (i.e. minimal) communication only once (assuming an optimizing compiler that eliminates common subexpressions), unless the array element may have been changed since its last use. For example, consider the code below:

```
      REAL a(1000), b(1000), c(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: a, b, c
      ...
      a(i) = b(i+2)                       ! Statement 1
      b(i) = c(i+3)                       ! Statement 2
      b(i+2) = 2 * a(i+2)                 ! Statement 3
      c(i) = a(i+1) + b(i+2) + c(i+3)     ! Statement 4
```

Statements 1 and 2 each require one array element to be communicated for any value of i. Statement 3 has no inherent communication. To simplify the discussion, assume that all four statements are executed on the processor storing the array element being assigned. [1] Then, for Statement 4:

- Element a(i+1) induces communication, since it is not local and was not communicated earlier;
- Element b(i+2) induces communication, since it is nonlocal and has changed since its last use; and
- Element c(i+3) *does not* induce new communication, since it was used in statement 2 and not changed since.

Thus, the minimum total inherent communication in this program fragment is four array elements. It is important to note that this is a minimum. Some compilation strategies may produce communication for element c(i+3) in the last statement.

2.2.2 Aggregate Communication Examples

The following examples illustrate the communication implications of some more complex constructs. The purpose is to show how communication can be quantified, but again the explanations do not necessarily reflect the actual compilation process. It is important to note that the communication requirement for each statement in this section is estimated without considering the surrounding context.

Consider the following statements:

[1] This is an optimal strategy for this example, although not for all programs.

```
      REAL a(1000), b(1000), c(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: c
      ...
      FORALL ( i = 1:1000 ) a(i) = b(i)       ! Forall 1
      FORALL ( i = 1:1000 ) a(i) = c(i)       ! Forall 2

      ! Forall 3
      FORALL ( i = 2:999 ) a(i) = a(i-1) + a(i) + a(i+1)

      ! Forall 4
      FORALL ( i = 2:999 ) c(i) = c(i-1) + c(i) + c(i+1)
```

The `FORALL` statement conceptually evaluates its right-hand side for all values of its indexes, then assigns to the left-hand side for all index values. These semantics allow parallel execution. Section 4 describes the `FORALL` statement in detail. The aggregate communication requirements of these statements follow directly from the inherent communication of the corresponding examples in Section 2.2.1.

In Forall 1, there is no inherent communication for any value of `i`; therefore, there is no communication for the aggregate construct.

In Forall 2, 90% of the references to `c(i)` are mapped to a processor different from that containing the corresponding `a(i)`. The aggregate communication must therefore transfer 900 array elements. Furthermore, analysis based on the definitions of `BLOCK` and `CYCLIC` shows that to update the values of `a` owned locally, each processor requires data from every other processor. For example, `procs(1)` must somehow receive:

- Elements $\{2, 12, 22, \ldots, 92\}$ from `procs(2)`;
- Elements $\{3, 13, 23, \ldots, 93\}$ from `procs(3)`; and
- So on for the other processors.

This produces an all-to-all communication pattern similar to the pattern for transposing a 2-dimensional array with certain distributions. The details of implementing such a pattern are very machine dependent and beyond the scope of this standard.

In Forall 3, the array references are all mapped to the same processor except for the first and last values of `i` on each processor. The aggregate communication requirement is therefore two array elements per processor (except `procs(1)` and `procs(10)`), or 18 elements total. Each processor must receive values from its left and right neighbors (again, except for `procs(1)` and `procs(10)`). This leads to a simple shift communication pattern (without wraparound).

In Forall 4, the update of each array element requires two off-processor values, each from a different processor. The total communication volume is therefore 1996 array elements. Further analysis reveals that all elements on processor `procs(k)` require elements from `procs(k` $\ominus$ `1)` and `procs(k` $\oplus$ `1)` (`MODULO(k - 2, 10) + 1` and `MODULO(k, 10) + 1` respectively, so called "clock arithmetic"). This leads to a massive shift communication pattern (with wraparound).

The aggregate communication for other constructs can be computed similarly. Iterative constructs generate the sum of the inherent communication for nested statements, while conditionals require at least the communication needed by the conditional branch that is taken. Repeated communication of the same array elements in any construct is not necessary unless the values of those elements may change.

Array expressions require an analysis similar to that for FORALL statements. In these cases, the inherent communication for each element of the result can be analyzed and the aggregate formed on that basis. The following statements have the same communication requirements as the above FORALL statements:

```
      REAL a(1000), b(1000), c(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs :: c
      ...
      ! Assignment 1 (equivalent to Forall 1)
      a(:) = b(:)

      ! Assignment 2 (equivalent to Forall 2)
      a(1:1000) = c(1:1000)

      ! Assignment 3 (equivalent to Forall 3)
      a(2:999) = a(1:998) + a(2:999) + a(3:1000)

      ! Assignment 4 (equivalent to Forall 4)
      c(2:999) = c(1:998) + c(2:999) + c(3:1000)
```

Some array intrinsics have inherent communication costs as well. For example, consider:

```
      REAL a(1000), b(1000), scalar
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b
      ...
      ! Intrinsic 1
      scalar = SUM( a )

      ! Intrinsic 2
      a = SPREAD( b(1), DIM=1, NCOPIES=1000 )

      ! Intrinsic 3
      a = CSHIFT(a,-1) + a + CSHIFT(a,1)
```

In general, the inherent communication derives from the mathematical definition of the function. For example, the inherent communication for computing SUM is one element for each processor storing part of the operand, minus one. (Further communication may be needed to store the result.) The optimal communication pattern is very machine-specific. Similar remarks apply to any accumulation operation; prefix and suffix intrinsics may require a larger volume based on the distribution. The SPREAD operation above requires a broadcast from procs(1) to all processors, which may take advantage of available hardware. The CSHIFT operations produce a shift communication pattern (with wraparound). This list of examples illustrating array intrinsics is not meant to be exhaustive.

There are other examples of situations in which nonaligned data must be communicated:

```
      REAL a(1000), c(100,100), d(100,100)
!HPF$ PROCESSORS procs(10)
!HPF$ ALIGN c(i,j) WITH d(j,i)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a
!HPF$ DISTRIBUTE (BLOCK,*) ONTO procs :: d
      ...
      a(1:200) = a(1:200) + a(2:400:2)
      c = c + d
```

In the first assignment, the use of different strides in the two references to `a` on the right-hand side will cause communication. The second assignment statement requires either a transpose of `c` or `d` or some complex communication pattern overlapping computation and communication.

A `REALIGN` directive may change the location of every element of the array. This will cause communication of all elements that change their home processor; in some compilation schemes, data will also be moved to new locations on the same processor. The communication volume is the same as an array assignment from an array with the original alignment to another array with the new alignment. The `REDISTRIBUTE` statement changes the distribution for every array aligned to the operand of the `REDISTRIBUTE`. Therefore, its cost is similar to the cost of a `REALIGN` on many arrays simultaneously. Compiler analysis may sometimes detect that data movement is not needed because an array has no values that could be accessed; such analysis and the resulting optimizations are beyond the scope of this document.

2.2.3 Interaction of Communication and Parallelism

The examples in Sections 2.2.1 and 2.2.2 were chosen so that parallelism and communication were not in conflict. The purpose of this section is to show cases where there is a tradeoff. The best implementation of all these examples will be machine dependent. As in the other sections, these examples do not necessarily reflect good programming practice.

Analyzing communication as in Sections 2.2.1 and 2.2.2 does not completely determine a program's performance. Consider the code:

```
      REAL x(100), y(100)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs:: x, y
      ...
      DO k = 3, 98
         x(k) = y(k) * (x(k-1) + x(k) + x(k+1)) / 3.0
         y(k) = x(k) + (y(k-1) + y(k-2) + y(k+1) + y(k+2)) / 4.0
      ENDDO
```

Only a few values need be communicated at the boundary of each processor. However, every iteration of the `DO` loop uses data computed on previous iterations for the references `x(k-1)`, `y(k-1)`, and `y(k-2)`. Therefore, although there is little inherent communication, the computation will run sequentially.

In contrast, consider the following code:

```
      REAL x(100), y(100), z(100)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs:: x, y, z
      ...
!HPF$ INDEPENDENT
      DO k = 3, 98
         x(k) = y(k) * (z(k-1) + z(k) + z(k+1)) / 3.0
         y(k) = x(k) + (z(k-1) + z(k-2) + z(k+1) + z(k+2)) / 4.0
      ENDDO
```

The `INDEPENDENT` directive asserts to the compiler that the iterations of the `DO` loop are completely independent of each other and none of the data accessed in the loop by an iteration is written by any other iteration.[2] Therefore, the loop has substantial potential parallelism and is likely to execute much faster than the last example. Section 4 describes the `INDEPENDENT` directive in more detail.

Assignment of work to processors may itself require communication. Consider the following code:

```
      INTEGER indx(1000), inv(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: indx, inv
      ...
      FORALL ( j = 1:1000 ) inv(indx(j)) = j**2
```

(Here, `indx` must be a permutation of the integers from 1 to 1000 in order for the FORALL to be well-defined.) Since the processor owning element `inv(indx(j))` depends on the values stored in `indx`, some data must be communicated simply to determine where the results will be stored. Two possible implementations of this are:

- Each processor calculates the squares for elements of `indx` that it owns and performs a scatter operation to communicate those values to the elements of `inv` where the final results are stored.

- Each processor determines the owner of `inv(indx(j))` for all elements of `indx` that it owns and notifies those processors. Each processor then computes the right-hand side for all elements for which it received notification.

In either case, nontrivial communication must be performed to distribute the work among processors. The optimal sharing scheme, its implementation, and its cost will be highly architecture dependent.

The parallelism in a section of code may conflict with the distribution of data, thus limiting the overall performance. Consider the following code:

```
      REAL a(1000,1000), b(1000,1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK,*) ONTO procs :: a, b
      ...
      DO i = 2, 1000
         a(i,:) = a(i,:) - (b(i,:)**2)/a(i-1,:)
      ENDDO
```

[2] Many compilers would detect this without the assertion. What cases of implicit parallelism are detected is highly compiler dependent and beyond the scope of this document.

Here, each iteration of the DO loop has a potential parallelism of 1000. However, all elements of a(i,:) and b(i,:) are located on the same processor. Therefore, exploitation of any of the potential parallelism will require scattering the data to other processors. (This is independent of the inherent communication required for the reference to a(i-1,:).) There are several implementation strategies available for the overall computation.

- Redistribute a and b before the DO loop to achieve the effect of

  ```
  !HPF$ DISTRIBUTE (*,BLOCK) ONTO procs :: a, b
  ```

  Redistribute back to the original distributions after the DO loop. This allows parallel updates of columns of a, at the cost of two all-to-all communication operations.

- Group the columns of a into blocks, then operate on the blocks separately. This strategy can produce a pipelined effect, allowing substantial parallelism. It sends many small messages to the neighboring processor rather than one large message.

- Execute the vector operations sequentially. This results in totally sequential operation, but avoids overhead from process start-up and small messages.

This list is not exhaustive. The optimal strategy will be highly machine dependent.

There is often a choice regarding where the result of an intermediate array expression will be stored, and different choices may lead to different communication performance. A straightforward implementation of the following code, for example, would require two transposition (communication) operations:

```
      REAL, DIMENSION(100,100) :: x, y, z
!HPF$ ALIGN WITH x :: y, z
      ...
      x = TRANSPOSE(y) + TRANSPOSE(z) + x
```

Despite two occurrences of the TRANSPOSE intrinsic, an optimizing compiler might implement this as:

```
      REAL, DIMENSION(100,100) :: x, y, z, t1
!HPF$ ALIGN WITH x :: y, z, t1
      ...
      t1 = y + z
      x = TRANSPOSE(t1) + x
```

with only one use of transposition.

Choosing an intermediate storage location is sometimes more complex, however. Consider the following code:

```
      REAL a(1000), b(1000), c(1000), d(1000)
      INTEGER ix(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (CYCLIC) ONTO procs:: a, b, c, d, ix
      ...
      a = b(ix) + c(ix) + d(ix)
```

and the following implementation strategies:

- Evaluate each element of the right-hand side on the processor where it will be stored. This strategy potentially requires fetching three values (the elements of `b`, `c`, and `d`) for each element computed. It always uses the maximum parallelism of the machine.
- Evaluate each element of the right-hand side on the processor where the corresponding elements of `b(ix)`, `c(ix)`, and `d(ix)` are stored. Ignoring set-up costs, this potentially communicates one result for each element computed. If the values of `ix` are evenly distributed, then it also uses the maximum machine parallelism.

On the basis of communication, the second strategy is better by a factor of 3; adding additional terms can make this factor arbitrarily large. However, that analysis does not consider parallel execution costs. If there are repeated values in `ix`, the second strategy may produce poor load balance. (For example, consider the case of `ix(i) = 10` for all `i`.) Minimizing this cost is a compiler optimization and is outside the scope of this language specification.

2.3 Syntax of Directives

HPF directives are consistent with Fortran 90 syntax in the following sense: if any HPF directive were to be adopted as part of a future Fortran standard, the only change necessary to convert an HPF program would be to replace the directive-origin with blanks.

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	*dynamic-directive*
		or	*inherit-directive*
		or	*template-directive*
		or	*combined-directive*
		or	*sequence-directive*
H205	*executable-directive*	**is**	*realign-directive*
		or	*redistribute-directive*
		or	*independent-directive*

Constraint: An *hpf-directive-line* cannot be commentary following another statement on the same line.

Constraint: A *specification-directive* may appear only where a *declaration-construct* may appear.

Constraint: An *executable-directive* may appear only where an *executable-construct* may appear.

Constraint: An *hpf-directive-line* follows the rules of either Fortran 90 free form (3.3.1.1) or fixed form (3.3.2.1) comment lines, depending on the source form of the surrounding Fortran 90 source form in that program unit. (3.3)

An *hpf-directive* is case insensitive and conforms to the rules for blanks in free source form (3.3.1), even in an HPF program otherwise in fixed source form. However an HPF-conforming processor is not required to diagnose extra or missing blanks in an HPF directive. Note that, due to Fortran 90 rules, the *directive-origin* in free source form must be the characters `!HPF$`. HPF directives may be continued, in which case each continued line also begins with a *directive-origin*. No statements may be interspersed within a continued HPF-directive. HPF directive lines must not appear within a continued statement. HPF directive lines may include trailing commentary.

In either source form, the blanks in the adjacent keywords `END FORALL` and `NO SEQUENCE` are optional.

An example of an HPF directive continuation in free source form is:

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

An example of an HPF directive continuation in fixed source form follows. Observe that column 6 must be blank, except when signifying continuation.

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

This example shows an HPF directive continuation which is "universal" in that it can be treated as either fixed source form or free source form. Note that the "&" in the first line is in column 73.

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

Section 3

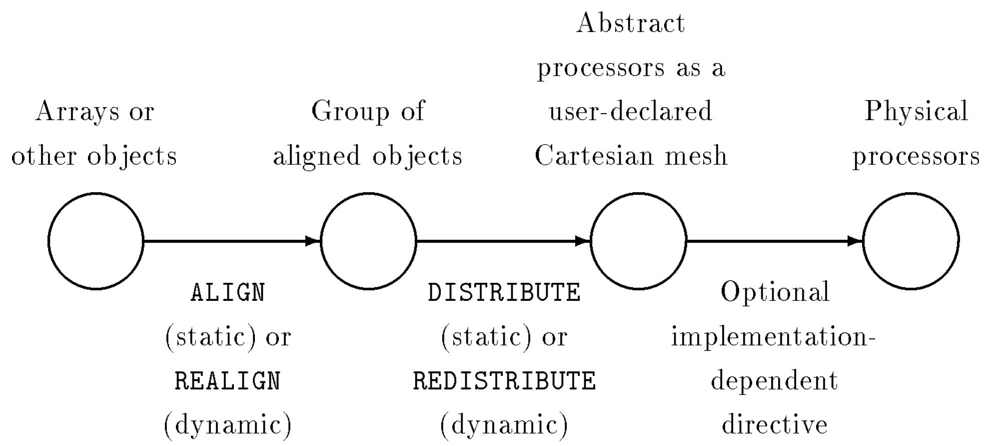
Data Alignment and Distribution Directives

HPF data alignment and distributions directives allow the programmer to advise the compiler how to assign array elements to processor memories.

3.1 Model

HPF adds directives to Fortran 90 to allow the user to advise the compiler on the allocation of data objects to processor memories. The model is that there is a two-level mapping of data objects to memory regions, referred to as "abstract processors." Data objects (typically array elements) are first *aligned* relative to one another; this group of arrays is then *distributed* onto a rectilinear arrangement of abstract processors. (The implementation then uses the same number, or perhaps some smaller number, of physical processors to implement these abstract processors. This mapping of abstract processors to physical processors is implementation-dependent.)

The following diagram illustrates the model:



The underlying assumptions are that an operation on two or more data objects is likely to be carried out much faster if they all reside in the same processor, and that it may be possible to carry out many such operations concurrently if they can be performed on different processors.

Fortran 90 provides a number of features, notably array syntax, that make it easy for a compiler to determine that many operations may be carried out concurrently. The HPF directives provide a way to inform the compiler of the recommendation that certain data objects should reside in the same processor: if two data objects are mapped (via the two-level mapping of alignment and distribution) to the same abstract processor, it is a strong recommendation to the implementation that they ought to reside in the same physical processor. There is also a provision for recommending that a data object be stored in multiple locations, which may complicate any updating of the object but makes it faster for multiple processors to read the object.

There is a clear separation between directives that serve as specification statements and directives that serve as executable statements (in the sense of the Fortran standards). Specification statements are carried out on entry to a program unit, as if all at once; only then are executable statements carried out. (While it is often convenient to think of specification statements as being handled at compile time, some of them contain specification expressions, which are permitted to depend on run-time quantities such as dummy arguments, and so the values of these expressions may not be available until run time, specifically the very moment that program control enters the scoping unit.)

The basic concept is that every array (indeed, every object) is created with *some* alignment to an entity, which in turn has *some* distribution onto *some* arrangement of abstract processors. If the specification statements contain explicit specification directives specifying the alignment of an array `A` with respect to another array `B`, then the distribution of `A` will be dictated by the distribution of `B`; otherwise, the distribution of `A` itself may be specified explicitly. In either case, any such explicit declarative information is used when the array is created.

> *Advice to implementors.* This model gives a better picture of the actual amount of work that needs to be done than a model that says "the array is created in some default location, and then realigned and/or redistributed if there is an explicit directive." Using `ALIGN` and `DISTRIBUTE` specification directives doesn't have to cause any more work at run time than using the implementation defaults. (*End of advice to implementors.*)

In the case of an allocatable object, we say that the object is created whenever it is allocated. Specification directives for allocatable objects (and allocated pointer targets) may appear in the *specification-part* of a program unit, but take effect each time the array is created, rather than on entry to the scoping unit.

Alignment is considered an *attribute* (in the Fortran 90 sense) of a data object. If an object `A` is aligned (statically or dynamically) with an object `B`, which in turn is already aligned to an object `C`, this is regarded as an alignment of `A` with `C` directly, with `B` serving only as an intermediary at the time of specification. (This matters only in the case where `B` is subsequently realigned; the result is that `A` remains aligned with `C`.) We say that `A` is *immediately aligned* with `B` but *ultimately aligned* with `C`. If an object is not explicitly aligned with another object, we say that it is ultimately aligned with itself. The alignment relationships form a tree with everything ultimately aligned to the object at the root of the tree; however, the tree is always immediately "collapsed" so that every object is related directly to the root. Any object that is not a root can be explicitly realigned but not explicitly redistributed. Any object that is a root can be explicitly redistributed but must not be explicitly realigned if anything else is aligned to it.

Every object which is the root of an alignment tree has an associated *template* or index space. Typically, this template has the same rank and size in each dimension as the object associated with it. (The most important exception to this rule is dummy arguments with the `INHERIT` attribute, described in Section 3.9.) We often refer to "the template for an array," which means the template of the object to which the array is ultimately aligned. (When an explicit `TEMPLATE` (see Section 3.8) is used, this may be simply the template to which the array is explicitly aligned.)

The *distribution* step of the HPF model technically applies to the template of an array, although because of the close relationship noted above we often speak loosely of the distribution of an array. Distribution partitions the template among a set of abstract processors according to a given pattern. The combination of alignment (from arrays to templates) and distribution (from templates to processors) thus determines the relationship of an array to the processors; we refer to this relationship as the *mapping* of the array. (These remarks also apply to a scalar, which may be regarded as having an index space whose sole position is indicated by an empty list of subscripts.)

Every object is created as if according to some complete set of specification directives; if the program does not include complete specifications for the mapping of some object, the compiler provides defaults. By default an object is not aligned with any other object; it is ultimately aligned with itself. The default distribution is implementation-dependent, but must be expressible as explicit directives for that implementation. (The distribution of a sequential object must be expressible as explicit directives only if it is an aggregate cover (see Section 7).) Identically declared objects need not be provided with identical default distribution specifications; the compiler may, for example, take into account the contexts in which objects are used in executable code. The programmer may force identically declared objects to have identical distributions by specifying such distributions explicitly. (On the other hand, identically declared processor arrangements *are* guaranteed to represent "the same processors arranged the same way." This is discussed in more detail in Section 3.7.)

Once an object has been created, it can be remapped by realigning it or redistributing an object to which it is ultimately aligned; but communication may be required in moving the data around. Redistributing an object causes all objects then ultimately aligned with it also to be redistributed so as to maintain the alignment relationships.

Sometimes it is desirable to consider a large index space with which several smaller arrays are to be aligned, but not to declare any array that spans the entire index space. HPF allows one to declare a `TEMPLATE`, which is like an array whose elements have no content and therefore occupy no storage; it is merely an abstract index space that can be distributed and with which arrays may be aligned.

By analogy with the Fortran 90 `ALLOCATABLE` attribute, HPF includes the attribute `DYNAMIC`. It is not permitted to `REALIGN` an array that has not been declared `DYNAMIC`. Similarly, it is not permitted to `REDISTRIBUTE` an array or template that has not been declared `DYNAMIC`.

3.2 Syntax of Data Alignment and Distribution Directives

Specification directives in HPF have two forms: specification statements, analogous to the `DIMENSION` and `ALLOCATABLE` statements of Fortran 90; and an attribute form analogous to type declaration statements in Fortran 90 using the "`::`" punctuation.

The attribute form allows more than one attribute to be described in a single directive. HPF goes beyond Fortran 90 in not requiring that the first attribute, or indeed any of them, be a type specifier.

For syntactic convenience, the executable directives `REALIGN` and `REDISTRIBUTE` also come in two forms (statement form and attribute form) but may not be combined with other attributes in a single directive.

H301	*combined-directive*	**is**	*combined-attribute-list* :: *entity-decl-list*
H302	*combined-attribute*	**is**	`ALIGN` *align-attribute-stuff*
		or	`DISTRIBUTE` *dist-attribute-stuff*
		or	`DYNAMIC`
		or	`INHERIT`
		or	`TEMPLATE`
		or	`PROCESSORS`
		or	`DIMENSION` (*explicit-shape-spec-list*)

Constraint: The same *combined-attribute* must not appear more than once in a given *combined-directive*.

Constraint: If the `DIMENSION` attribute appears in a *combined-directive*, any entity to which it applies must be declared with the HPF `TEMPLATE` or `PROCESSORS` type specifier.

The following rules constrain the declaration of various attributes, whether in separate directives or in a combined-directive.

If the `DISTRIBUTE` attribute is present, then every name declared in the *entity-decl-list* is considered to be a *distributee* and is subject to the constraints listed in section 3.3.

If the `ALIGN` attribute is present, then every name declared in the *entity-decl-list* is considered to be an *alignee* and is subject to the constraints listed in section 3.4.

The HPF keywords `PROCESSORS` and `TEMPLATE` play the role of type specifiers in declaring processor arrangements and templates. The HPF keywords `ALIGN`, `DISTRIBUTE`, `DYNAMIC`, and `INHERIT` play the role of attributes. Attributes referring to processor arrangements, to templates, or to entities with other types (such as `REAL`) may be combined in an HPF directive without having the type specifier appear.

No entity may be given a particular attribute more than once.

Dimension information may be specified after an *object-name* or in a `DIMENSION` attribute. If both are present, the one after the *object-name* overrides the `DIMENSION` attribute (this is consistent with the Fortran 90 standard). For example, in:

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

`A`, `B`, and `D` are 64×64 templates; `C` is 32×32.

If a specification expression includes a reference to the value of an element of an array specified in the same specification-part, any explicit mapping or `INHERIT` attribute for the array must be completely specified in prior specification-directives. (This restriction is inspired by and extends section 7.1.6.2 of the Fortran 90 standard, which states in part: If a specification expression includes a reference to the value of an element of an array specified in the same specification-part, the array bounds must be specified in a prior declaration.

A comment on asterisks: The asterisk character "*" appears in the syntax rules for HPF alignment and distribution directives in three distinct roles:

- When a lone asterisk appears as a member of a parenthesized list, it indicates either a collapsed mapping, wherein many elements of an array may be mapped to the same abstract processor, or a replicated mapping, wherein each element of an array may be mapped to many abstract processors. See the syntax rules for *align-source* and *align-subscript* (see Section 3.4) and for *dist-format* (see Section 3.3).

- When an asterisk appears before a left parenthesis "(" or after the keyword `WITH` or `ONTO`, it indicates that the directive constitutes an assertion about the *current* mapping of a dummy argument on entry to a subprogram, rather than a request for a *desired* mapping of that dummy argument. This use of the asterisk may appear *only* in directives that apply to dummy arguments (see Section 3.10).

- When an asterisk appears in an *align-subscript-use* expression, it represents the usual integer multiplication operator.

3.3 DISTRIBUTE and REDISTRIBUTE Directives

The `DISTRIBUTE` directive specifies a mapping of data objects to abstract processors in a processor arrangement. For example,

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

specifies that the array `SALAMI` should be distributed across some set of abstract processors by slicing it uniformly into blocks of contiguous elements. If there are 50 processors, the directive implies that the array should be divided into groups of 200 elements, with `SALAMI(1:200)` mapped to the first processor, `SALAMI(201:400)` mapped to the second processor, and so on. If there is only one processor, the entire array is mapped to that processor as a single block of 10000 elements.

The block size may be specified explicitly:

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

This specifies that groups of exactly 256 elements should be mapped to successive abstract processors. (There must be at least $\lceil 10000/256 \rceil = 40$ abstract processors if the directive is to be satisfied. The fortieth processor will contain a partial block of only 16 elements, namely `WEISSWURST(9985:10000)`.)

HPF also provides a cyclic distribution format:

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

If there are 4 abstract processors, the first processor will contain `DECK_OF_CARDS(1:49:4)`, the second processor will contain `DECK_OF_CARDS(2:50:4)`, the third processor will contain `DECK_OF_CARDS(3:51:4)`, and the fourth processor will contain `DECK_OF_CARDS(4:52:4)`. Successive array elements are dealt out to successive abstract processors in round-robin fashion.

Distributions may be specified independently for each dimension of a multidimensional array:

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

The CHESS_BOARD array will be carved up into contiguous rectangular patches, which will be distributed onto a two-dimensional arrangement of abstract processors. The GO_BOARD array will have its rows distributed cyclically over a one-dimensional arrangement of abstract processors. (The "*" specifies that GO_BOARD is not to be distributed along its second axis; thus an entire row is to be distributed as one object. This is sometimes called "on-processor" distribution.)

The REDISTRIBUTE directive is similar to the DISTRIBUTE directive but is considered executable. An array (or template) may be redistributed at any time, provided it has been declared DYNAMIC (see Section 3.5). Any other arrays currently ultimately aligned with an array (or template) when it is redistributed are also remapped to reflect the new distribution, in such a way as to preserve alignment relationships (see Section 3.4). (This can require a lot of computational and communication effort at run time; the programmer must take care when using this feature.)

The DISTRIBUTE directive may appear only in the *specification-part* of a scoping unit. The REDISTRIBUTE directive may appear only in the *execution-part* of a scoping unit. The principal difference between DISTRIBUTE and REDISTRIBUTE is that DISTRIBUTE must contain only a *specification-expr* as the argument to a BLOCK or CYCLIC option, whereas in REDISTRIBUTE such an argument may be any integer expression. Another difference is that DISTRIBUTE is an attribute, and so can be combined with other attributes as part of a *combined-directive*, whereas REDISTRIBUTE is not an attribute (although a REDISTRIBUTE statement may be written in the style of attributed syntax, using "::" punctuation).

Formally, the syntax of the DISTRIBUTE and REDISTRIBUTE directives is:

H303	*distribute-directive*	**is**	DISTRIBUTE *distributee dist-directive-stuff*
H304	*redistribute-directive*	**is**	REDISTRIBUTE *distributee dist-directive-stuff*
		or	REDISTRIBUTE *dist-attribute-stuff* :: *distributee-list*
H305	*dist-directive-stuff*	**is**	*dist-format-clause* [*dist-onto-clause*]
H306	*dist-attribute-stuff*	**is**	*dist-directive-stuff*
		or	*dist-onto-clause*
H307	*distributee*	**is**	*object-name*
		or	*template-name*
H308	*dist-format-clause*	**is**	(*dist-format-list*)
		or	* (*dist-format-list*)
		or	*
H309	*dist-format*	**is**	BLOCK [(*int-expr*)]
		or	CYCLIC [(*int-expr*)]
		or	*
H310	*dist-onto-clause*	**is**	ONTO *dist-target*
H311	*dist-target*	**is**	*processors-name*
		or	* *processors-name*
		or	*

Constraint: An *object-name* mentioned as a *distributee* must be a simple name and not a subobject designator.

Constraint: An *object-name* mentioned as a *distributee* may not appear as an *alignee*.

Constraint: An *object-name* mentioned as a *distributee* may not have the `POINTER` attribute.

Constraint: A *distributee* that appears in a `REDISTRIBUTE` directive must have the `DYNAMIC` attribute (see Section 3.5).

Constraint: If a *dist-format-list* is specified, its length must equal the rank of each *distributee*.

Constraint: If both a *dist-format-list* and a *processors-name* appear, the number of elements of the *dist-format-list* that are not "*" must equal the rank of the named processor arrangement.

Constraint: If a *processors-name* appears but not a *dist-format-list*, the rank of each *distributee* must equal the rank of the named processor arrangement.

Constraint: If either the *dist-format-clause* or the *dist-target* in a `DISTRIBUTE` directive begins with "*" then every *distributee* must be a dummy argument.

Constraint: Neither the *dist-format-clause* nor the *dist-target* in a `REDISTRIBUTE` may begin with "*".

Constraint: Any *int-expr* appearing in a *dist-format* of a `DISTRIBUTE` directive must be a *specification-expr*.

Note that the possibility of a `DISTRIBUTE` directive of the form

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

is covered by syntax rule H301 for a *combined-directive*.

Examples:

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

The meanings of the alternatives for *dist-format* are given below.

Define the ceiling division function `CD(J,K) = (J+K-1)/K` (using Fortran integer arithmetic with truncation toward zero.)

Define the ceiling remainder function `CR(J,K) = J-K*CD(J,K)`.

The dimensions of a processor arrangement appearing as a *dist-target* are said to *correspond* in left-to-right order with those dimensions of a *distributee* for which the corresponding *dist-format* is not `*`. In the example above, processor arrangement `SQUARE` must be two-dimensional; its first dimension corresponds to the first dimensions of `D2`, `D3`, and `D4` and its second dimension corresponds to the third dimensions of `D2`, `D3`, and `D4`.

Let d be the size of a *distributee* in a certain dimension and let p be the size of the processor arrangement in the corresponding dimension. For simplicity, assume all dimensions have a lower bound of 1. Then `BLOCK`(m) means that a *distributee* position whose index along that dimension is j is mapped to an abstract processor whose index along the corresponding dimension of the processor arrangement is `CD`(j,m) (note that $m \times p \geq d$ must be true), and is position number m+CR(j,m) among positions mapped to that abstract processor. The first *distributee* position in abstract processor k along that axis is position number 1+m*(k-1).

The block size m must be a positive integer.

BLOCK by definition means the same as BLOCK(CD(d,p)).

CYCLIC(m) means that a *distributee* position whose index along that dimension is j is mapped to an abstract processor whose index along the corresponding dimension of the processor arrangement is 1+MODULO(CD(j,m)-1,p). The first *distributee* position in abstract processor k along that axis is position number 1+m*(k-1).

The block size m must be a positive integer.

CYCLIC by definition means the same as CYCLIC(1).

CYCLIC(m) and BLOCK(m) imply the same distribution when $m \times p \geq d$, but BLOCK(m) additionally asserts that the distribution will not wrap around in a cyclic manner, which a compiler cannot determine at compile time if m is not constant. Note that CYCLIC and BLOCK (without argument expressions) do not imply the same distribution unless $p \geq d$, a degenerate case in which the block size is 1 and the distribution does not wrap around.

Suppose that we have 16 abstract processors and an array of length 100:

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

Distributing the array BLOCK (which in this case would mean the same as BLOCK(7)):

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

results in this mapping of array elements onto abstract processors:

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

Distributing the array BLOCK(8):

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

results in this mapping of array elements onto abstract processors:

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

Distributing the array `BLOCK(6)` is not HPF-conforming because $6 \times 16 < 100$.

Distributing the array `CYCLIC` (which means exactly the same as `CYCLIC(1)`):

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

results in this mapping of array elements onto abstract processors:

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

Distributing the array `CYCLIC(3)`:

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

results in this mapping of array elements onto abstract processors:

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															

Note that it is perfectly permissible for an array to be distributed so that some processors have no elements. Indeed, an array may be "distributed" so that all elements reside on one processor. For example,

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

results in having only one non-empty block—a partially-filled one at that, having only 100 elements—on processor 1, with processors 2 through 16 having no elements of the array.

A `DISTRIBUTE` or `REDISTRIBUTE` directive must not cause any data object associated with the *distributee* via storage association (`COMMON` or `EQUIVALENCE`) to be mapped such that storage units of a scalar data object are split across more than one abstract processor. See Section 7 for further discussion of storage association.

The statement form of a `DISTRIBUTE` or `REDISTRIBUTE` directive may be considered an abbreviation for an attributed form that happens to mention only one *distributee*; for example,

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

is equivalent to

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

Note that, to prevent syntactic ambiguity, the *dist-format-clause* must be present in the statement form, so in general the statement form of the directive may not be used to specify the mapping of scalars.

If the *dist-format-clause* is omitted from the attributed form, then the language processor may make an arbitrary choice of distribution formats for each template or array. So the directive

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

means the same as

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

to which a compiler, perhaps taking into account patterns of use of `D1`, `D2`, and `D3` within the code, might choose to supply three distinct distributions such as, for example,

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

Then again, the compiler might happen to choose the same distribution for all three arrays.

In either the statement form or the attributed form, if the `ONTO` clause is present, it specifies the processor arrangement that is the target of the distribution. If the `ONTO` clause is omitted, then a implementation-dependent processor arrangement is chosen arbitrarily for each *distributee*. So, for example,

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

causes the arrays `ARTHUR` and `ARNOLD` to have the same mapping, so that corresponding elements reside in the same abstract processor, because they are the same size and distributed in the same way (`BLOCK`) onto the same processor arrangement (`EXCALIBUR`). However, `LUCY` and `LINUS` do not necessarily have the same mapping because they might, depending on the implementation, be distributed onto differently chosen processor arrangements; so corresponding elements of `LUCY` and `LINUS` might not reside on the same abstract processor. (The `ALIGN` directive provides a way to ensure that two arrays have the same mapping without having to specify an explicit processor arrangement.)

3.4 ALIGN and REALIGN Directives

The `ALIGN` directive is used to specify that certain data objects are to be mapped in the same way as certain other data objects. Operations between aligned data objects are likely to be more efficient than operations between data objects that are not known to be aligned (because two objects that are aligned are intended to be mapped to the same abstract processor). The `ALIGN` directive is designed to make it particularly easy to specify explicit mappings for all the elements of an array at once. While objects can be aligned in some cases through careful use of matching `DISTRIBUTE` directives, `ALIGN` is more general and frequently more convenient.

The `REALIGN` directive is similar to the `ALIGN` directive but is considered executable. An array (or template) may be realigned at any time, provided it has been declared `DYNAMIC` (see Section 3.5) Unlike redistribution (see Section 3.3), realigning a data object does not cause any other object to be remapped. (However, realignment of even a single object, if it is large, could require a lot of computational and communication effort at run time; the programmer must take care when using this feature.)

The `ALIGN` directive may appear only in the *specification-part* of a scoping unit. The `REALIGN` directive is similar but may appear only in the *execution-part* of a scoping unit. The principal difference between `ALIGN` and `REALIGN` is that `ALIGN` must contain only a *specification-expr* as a *subscript* or in a *subscript-triplet*, whereas in `REALIGN` such subscripts may be any integer expressions. Another difference is that `ALIGN` is an attribute, and so can be combined with other attributes as part of a *combined-directive*, whereas `REALIGN` is not an attribute (although a `REALIGN` statement may be written in the style of attributed syntax, using "`::`" punctuation).

Formally, the syntax of `ALIGN` and `REALIGN` is as follows:

H312	*align-directive*	**is**	`ALIGN` *alignee* *align-directive-stuff*
H313	*realign-directive*	**is**	`REALIGN` *alignee* *align-directive-stuff*
		or	`REALIGN` *align-attribute-stuff* `::` *alignee-list*
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*

Constraint: An *object-name* mentioned as an *alignee* must be a simple name and not a subobject designator.

Constraint: An *object-name* mentioned as an *alignee* may not appear as a *distributee*.

Constraint: An *object-name* mentioned as an *alignee* may not have the `POINTER` attribute.

Constraint: Any *alignee* that appears in a `REALIGN` directive must have the `DYNAMIC` attribute (see Section 3.5).

Constraint: If the `align-target` specified in the `align-with-clause` has the `DYNAMIC` attribute, then each `alignee` must also have the `DYNAMIC` attribute.

Constraint: If the *alignee* is scalar, the *align-source-list* (and its surrounding parentheses) must not appear. In this case the statement form of the directive is not allowed.

Constraint: If the *align-source-list* is present, its length must equal the rank of the alignee.

Constraint: An *align-dummy* must be a named variable.

Constraint: An object may not have both the `INHERIT` attribute and the `ALIGN` attribute. (However, an object with the `INHERIT` attribute may appear as an *alignee* in a `REALIGN` directive, provided that it does not appear as a *distributee* in a `DISTRIBUTE` or `REDISTRIBUTE` directive.)

Note that the possibility of an `ALIGN` directive of the form

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

is covered by syntax rule H301 for a *combined-directive*.

The statement form of an `ALIGN` or `REALIGN` directive may be considered an abbreviation of an attributed form that happens to mention only one *alignee*:

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

is equivalent to

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

If the *align-source-list* is omitted from the attributed form and the *alignee*s are not scalar, the *align-source-list* is assumed to consist of a parenthesized list of "`:`" entries, equal in number to the rank of the *alignees*. Similarly, if the *align-subscript-list* is omitted from the *align-spec* in either form, it is assumed to consist of a parenthesized list of "`:`" entries, equal in number to the rank of the *align-target*. So the directive

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

means

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

which in turn means the same as

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

because an attributed-form directive that mentions more than one *alignee* is equivalent to a series of identical directives, one for each *alignee*; all *alignee*s must have the same rank. With this understanding, we will assume below, for the sake of simplifying the description, that an `ALIGN` or `REALIGN` directive has a single *alignee*.

Each *align-source* corresponds to one axis of the *alignee*, and is specified as either "`:`" or "`*`" or a dummy variable:

- If it is "`:`", then positions along that axis will be spread out across the matching axis of the *align-spec* (see below).
- If it is "`*`", then that axis is *collapsed*: positions along that axis make no difference in determining the corresponding position within the *align-target*. (Replacing the "`*`" with a dummy variable name not used anywhere else in the directive would have the same effect; "`*`" is merely a convenience that saves the trouble of inventing a variable name and makes it clear that no dependence on that dimension is intended.)
- A dummy variable is considered to range over all valid index values for that dimension of the *alignee*.

The `WITH` clause of an `ALIGN` has the following syntax:

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*

Constraint: An *object-name* mentioned as an *align-target* must be a simple name and not a subobject designator.

Constraint: An *align-target* may not have the `OPTIONAL` attribute.

Constraint: If the *align-spec* in an `ALIGN` directive begins with "`*`" then every *alignee* must be a dummy argument.

Constraint: The *align-spec* in a `REALIGN` may not begin with "`*`".

Constraint: Each *align-dummy* may appear at most once in an *align-subscript-list*.

Constraint: An *align-subscript-use* expression may contain at most one occurrence of an *align-dummy*.

Constraint: An *align-dummy* may not appear anywhere in the *align-spec* except where explicitly permitted to appear by virtue of the grammar shown above. Paraphrased, one may construct an *align-subscript-use* by starting with an *align-dummy* and then doing additive and multiplicative things to it with any integer expressions that contain no *align-dummy*.

Constraint: A *subscript* in an *align-subscript* may not contain occurrences of any *align-dummy*.

Constraint: An *int-add-operand*, *int-mult-operand*, or *int-level-two-expr* must be of type integer.

The syntax rules for an *align-subscript-use* take account of operator precedence issues, but the basic idea is simple: an *align-subscript-use* is intended to be a linear function of a single occurrence of an *align-dummy*.

For example, the following *align-subscript-use* expressions are valid, assuming that `J`, `K`, and `M` are *align-dummy*s and `N` is not an *align-dummy*:

```
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
```

The following expressions are not valid *align-subscript-use* expressions:

```
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)
```

The *align-spec* must contain exactly as many *subscript-triplets* as the number of colons ("`:`") appearing in the *align-source-list*. These are matched up in corresponding left-to-right order, ignoring, for this purpose, any *align-source* that is not a colon and any *align-subscript* that is not a *subscript-triplet*. Consider a dimension of the *alignee* for which a colon appears as an *align-source* and let the lower and upper bounds of that array be LA and UA. Let the corresponding subscript triplet be LT:UT:ST or its equivalent. Then the colon could be replaced by a new, as-yet-unused dummy variable, say `J`, and the subscript triplet by the expression `(J-`LA`)*`ST`+`LT without affecting the meaning of the directive. Moreover, the axes must conform, which means that

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

must be true. (This is entirely analogous to the treatment of array assignment.)

To simplify the remainder of the discussion, we assume that every colon in the *align-source-list* has been replaced by new dummy variables in exactly the fashion just described, and that every "`*`" in the *align-source-list* has likewise been replaced by an otherwise unused dummy variable. For example,

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

may be transformed into its equivalent

```
!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)
```

with the attached requirements

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

Thus we need consider further only the case where every *align-source* is a dummy variable and no *align-subscript* is a *subscript-triplet*.

Each dummy variable is considered to range over all valid index values for the corresponding dimension of the *alignee*. Every combination of possible values for the index variables selects an element of the *alignee*. The *align-spec* indicates a corresponding element (or section) of the *align-target* with which that element of the *alignee* should be aligned; this indication may be a function of the index values, but the nature of this function is syntactically restricted (as discussed above) to linear functions in order to limit the complexity of the implementation. Each *align-dummy* variable may appear at most once in the *align-spec* and only in certain rigidly prescribed contexts. The result is that each *align-subscript* expression may contain at most one *align-dummy* variable and the expression is constrained to be a linear function of that variable. (Therefore skew alignments are not possible.)

An asterisk "*" as an *align-subscript* indicates a replicated representation. Each element of the *alignee* is aligned with every position along that axis of the *align-target*.

> *Rationale.* It may seem strange to use "*" to mean both collapsing and replication; the rationale is that "*" always stands conceptually for a dummy variable that appears nowhere else in the statement and ranges over the set of indices for the indicated dimension. Thus, for example,
>
> ```
> !HPF$ ALIGN A(:) WITH D(:,*)
> ```
>
> means that a copy of `A` is aligned with every column of `D`, because it is conceptually equivalent to
>
> *for every legitimate index j, align* `A(:)` *with* `D(:,`*j*`)`
>
> just as
>
> ```
> !HPF$ ALIGN A(:,*) WITH D(:)
> ```
>
> is conceptually equivalent to
>
> *for every legitimate index j, align* `A(:,`*j*`)` *with* `D(:)`
>
> Note, however, that while HPF syntax allows
>
> ```
> !HPF$ ALIGN A(:,*) WITH D(:)
> ```

to be written in the alternate form

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

it does *not* allow

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

to be written in the alternate form

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

because that has another meaning (only a variable appearing in the *align-source-list* following the *alignee* is understood to be an *align-dummy*, so the current value of the variable J is used, thus aligning A with a single column of D).

Replication allows an optimizing compiler to arrange to read whichever copy is closest. (Of course, when a replicated data object is written, all copies must be updated, not just one copy. Replicated representations are very useful for use as small lookup tables, where it is much faster to have a copy in each physical processor but without giving it an extra dimension that is logically unnecessary to the algorithm.) (*End of rationale.*)

By applying the transformations given above, all cases of an *align-subscript* may be conceptually reduced to either an *int-expr* (not involving an *align-dummy*) or an *align-subscript-use* and the *align-source-list* may be reduced to a list of index variables with no "*" or ":". An *align-subscript-list* may then be evaluated for any specific combination of values for the *align-dummy* variables simply by evaluating each *align-subscript* as an expression. The resulting subscript values must be legitimate subscripts for the *align-target*. (This implies that the *alignee* is not allowed to "wrap around" or "extend past the edges" of an *align-target*.) The selected element of the *alignee* is then considered to be aligned with the indicated element of the *align-target*; more precisely, the selected element of the *alignee* is considered to be ultimately aligned with the same object with which the indicated element of the *align-target* is currently ultimately aligned (possibly itself).

Once a relationship of ultimate alignment is established, it persists, even if the ultimate *align-target* is redistributed, unless and until the *alignee* is realigned by a REALIGN directive, which is permissible only if the *alignee* has the DYNAMIC attribute.

More examples of ALIGN directives:

```
      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, DYNAMIC:: P,Q,R,S
```

Note that, in a *alignee-list*, the alignees must all have the same rank but need not all have the same shape; the extents need match only for dimensions that correspond to colons in the *align-source-list*. This turns out to be an extremely important convenience; one of the most common cases in current practice is aligning arrays that match in distributed ("parallel") dimensions but may differ in collapsed ("on-processor") dimensions:

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

Here there are processors (perhaps N of them) and arrays of different sizes (3, 4, 43) within each processor are required. As far as HPF is concerned, the numbers 3, 4, and 43 may be different, because those axes will be collapsed. Thus array elements with indices differing only along that axis will all be aligned with the same element of Q (and thus be specified as residing in the same processor).

In the following examples, each directive in the group means the same thing, assuming that corresponding axis upper and lower bounds match:

```
!Second axis of X is collapsed
!HPF$ ALIGN X(:,*) WITH D1(:)
!HPF$ ALIGN X(J,*) WITH D1(J)
!HPF$ ALIGN X(J,K) WITH D1(J)

!Replicated representation along second axis of D3
!HPF$ ALIGN X(:,:) WITH D3(:,*,:)
!HPF$ ALIGN X(J,K) WITH D3(J,*,K)

!Transposing two axes
!HPF$ ALIGN X(J,K) WITH D2(K,J)
!HPF$ ALIGN X(J,:) WITH D2(:,J)
!HPF$ ALIGN X(:,K) WITH D2(K,:)
!But there isn't any way to get rid of *both* index variables;
! the subscript-triplet syntax alone cannot express transposition.

!Reversing both axes
!HPF$ ALIGN X(J,K) WITH D2(M-J+1,N-K+1)
!HPF$ ALIGN X(:,:) WITH D2(M:1:-1,N:1:-1)

!Simple case
!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 DYNAMIC Directive

The DYNAMIC attribute specifies that an object may be dynamically realigned or redistributed.

H329	*dynamic-directive*	**is**	DYNAMIC *alignee-or-distributee-list*
H330	*alignee-or-distributee*	**is**	*alignee*
		or	*distributee*

Constraint: An object in COMMON may not be declared DYNAMIC and may not be aligned to an object (or template) that is DYNAMIC. (To get this kind of effect, Fortran 90 modules must be used instead of COMMON blocks.)

Constraint: An object with the SAVE attribute may not be declared DYNAMIC and may not be aligned to an object (or template) that is DYNAMIC.

A REALIGN directive may not be applied to an *alignee* that does not have the DYNAMIC attribute. A REDISTRIBUTE directive may not be applied to a *distributee* that does not have the DYNAMIC attribute.

A DYNAMIC directive may be combined with other directives, with the attributes stated in any order, consistent with the Fortran 90 attribute syntax.

Examples:

```
!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
```

The first two examples mean exactly the same thing. The next two examples mean exactly the same second thing. The last two examples mean exactly the same third thing.

The three directives

```
!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
```

may be combined into a single directive as follows:

```
!HPF$ TEMPLATE, DISTRIBUTE(BLOCK, BLOCK) ONTO P,   &
!HPF$   DIMENSION(64,64),DYNAMIC :: A,B,C,D
```

3.6 Allocatable Arrays and Pointers

A variable with the POINTER or ALLOCATABLE attribute may appear as an *alignee* in an ALIGN directive or as a *distributee* in a DISTRIBUTE directive. Such directives do not take effect immediately, however; they take effect each time the array is allocated by an ALLOCATE statement, rather than on entry to the scoping unit. The values of all specification expressions in such a directive are determined once on entry to the scoping unit and may be used multiple times (or not at all). For example:

```
      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))
      ...
```

The values of the expressions N and M*2 on entry to the subprogram are conceptually retained by the ALIGN and DISTRIBUTE directives for later use at allocation time. When the array A is allocated, it is distributed with a block size equal to the retained value of M*2, not the value 182. When the array B is allocated, it is aligned relative to A according to the retained value of N, not its new value 43.

Note that it would have been incorrect in the MILLARD_FILLMORE example to perform the two ALLOCATE statements in the opposite order. In general, when an object X is created it may be aligned to another object Y only if Y has already been created or allocated. The following example illustrates several related cases.

```
      SUBROUTINE WARREN_HARDING(P,Q)
      REAL P(:)
      REAL Q(:)
      REAL R(SIZE(Q))
      REAL, ALLOCATABLE :: S(:),T(:)
!HPF$ ALIGN P(I) WITH T(I)                          !Nonconforming
!HPF$ ALIGN Q(I) WITH *T(I)                         !Nonconforming
!HPF$ ALIGN R(I) WITH T(I)                          !Nonconforming
!HPF$ ALIGN S(I) WITH T(I)
      ALLOCATE(S(SIZE(Q)))                          !Nonconforming
      ALLOCATE(T(SIZE(Q)))
```

The ALIGN directives are not HPF-conforming because the array T has not yet been allocated at the time that the various alignments must take place. The four cases differ slightly in their details. The arrays P and Q already exist on entry to the subroutine, but because T is not yet allocated, one cannot correctly prescribe the alignment of P or describe the alignment of Q relative to T. (See Section 3.10 for a discussion of prescriptive and descriptive directives.) The array R is created on subroutine entry and its size can correctly depend on the SIZE of Q, but the alignment of R cannot be specified in terms of the alignment of T any more than its size can be specified in terms of the size of T. It *is* permitted to have an alignment directive for S in terms of T, because the alignment action does not take place until S is allocated; however, the first ALLOCATE statement is nonconforming because S needs to be aligned but at that point in time T is still unallocated.

If an ALLOCATE statement is immediately followed by REDISTRIBUTE and/or REALIGN directives, the meaning in principle is that the array is first created with the statically declared alignment, then immediately remapped. In practice there is an obvious optimization: create the array in the processors to which it is about to be remapped, in a single step. HPF implementors are strongly encouraged to implement this optimization and HPF programmers are encouraged to rely upon it. Here is an example:

```
      REAL,ALLOCATABLE(:,:) :: TINKER, EVERS
!HPF$ DYNAMIC :: TINKER, EVERS
      REAL, POINTER :: 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)
```

While `CHANCE` is by default always allocated with a `BLOCK` distribution, it should be possible for a compiler to notice that it will immediately be remapped to a `CYCLIC` distribution. Similar remarks apply to `TINKER` and `EVERS`. (Note that `EVERS` is mapped in a thinly-spread-out manner onto `TINKER`; adjacent elements of `EVERS` are mapped to elements of `TINKER` separated by a stride `M`. This thinly-spread-out mapping is put in the lower left corner of `TINKER`, because `EVERS(1,1)` is mapped to `TINKER(M,1)`.)

An array pointer may be used in `REALIGN` and `REDISTRIBUTE` as an *alignee*, *align-target*, or *distributee* if and only if it is currently associated with a whole array, not an array section. One may remap an object by using a pointer as an *alignee* or *distributee* only if the object was created by `ALLOCATE` but is not an `ALLOCATABLE` array.

Any directive that remaps an object constitutes an assertion on the part of the programmer that the remainder of program execution would be unaffected if all pointers associated with any portion of the object were instantly to acquire undefined pointer association status, except for the one pointer, if any, used to indicate the object in the remapping directive.

> *Advice to implementors.* If HPF directives were ever to be absorbed as actual Fortran statements, the previous paragraph could be written as "Remapping an object causes all pointers associated with any portion of the object to have undefined pointer association status, except for the one pointer, if any, used to indicate the object in the remapping directive." The more complicated wording here is intended to avoid any implication that the remapping directives, in the form of structured comment annotations, have any effect on the execution semantics, as opposed to the execution speed, of the annotated program.) (*End of advice to implementors.*)

When an array is allocated, it will be aligned to an existing template if there is an explicit `ALIGN` directive for the allocatable variable. If there is no explicit `ALIGN` directive, then the array will be ultimately aligned with itself. It is forbidden for any other object to be ultimately aligned to an array at the time the array becomes undefined by reason of deallocation. All this applies regardless of whether the name originally used in the `ALLOCATE` statement when the array was created had the `ALLOCATABLE` attribute or the `POINTER` attribute.

3.7 PROCESSORS Directive

The `PROCESSORS` directive declares one or more rectilinear processor arrangements, specifying for each one its name, its rank (number of dimensions), and the extent in each dimension. It may appear only in the *specification-part* of a scoping unit. Every dimension of a processor arrangement must have nonzero extent; therefore a processor arrangement cannot be empty.

In the language of section 14.1.2 of the Fortran 90 standard, processor arrangements are local entities of class (1); therefore a processor arrangement may not have the same name as a variable, named constant, internal procedure, etc., in the same scoping unit. Names of processor arrangements obey the same rules for host and use association as other names in the long list in section 12.1.2.2.1 of the Fortran 90 standard.

A processor arrangement declared in a module has the default accessibility of the module.

Rationale. Because the name of a processor arrangement is not a first-class entity in HPF, but must appear only in directives, it cannot appear in an *access-stmt* (`PRIVATE` or `PUBLIC`). If directives ever become full-fledged Fortran statements rather than structured comments, then it would be appropriate to allow the accessibility of a processor arrangement to be controlled by listing its name in an *access-stmt*. (*End of rationale.*)

If two processor arrangements have the same shape, then corresponding elements of the two arrangements are understood to refer to the same abstract processor. (It is anticipated that implementation-dependent directives provided by some HPF implementations could overrule the default correspondence of processor arrangements that have the same shape.)

If directives collectively specify that two objects be mapped to the same abstract processor at a given instant during the program execution, the intent is that the two objects be mapped to the same physical processor at that instant.

The intrinsic functions `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE` may be used to inquire about the total number of actual physical processors used to execute the program. This information may then be used to calculate appropriate sizes for the declared abstract processor arrangements.

H331	*processors-directive*	**is**	`PROCESSORS` *processors-decl-list*
H332	*processors-decl*	**is**	*processors-name* [(*explicit-shape-spec-list*)]
H333	*processors-name*	**is**	*object-name*

Examples:

```
!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
```

If no shape is specified, then the declared processor arrangement is conceptually scalar.

Rationale. A scalar processor arrangement may be useful as a way of indicating that certain scalar data should be kept together but need not interact strongly with distributed data. Depending on the implementation architecture, data distributed onto such a processor arrangement may reside in a single "control" or "host" processor (if the machine has one), or may reside in an arbitrarily chosen processor, or may be replicated over all processors. For target architectures that have a set of computational processors and a separate scalar host computer, a natural implementation is to map every scalar processor arrangement onto the host processor. For target architectures that have a set of computational processors but no separate scalar "host" computer, data mapped to a scalar processor arrangement might be mapped to some arbitrarily chosen computational processor or replicated onto all computational processors. (*End of rationale.*)

An HPF compiler is required to accept any PROCESSORS declaration in which the product of the extents of each declared processor arrangement is equal to the number of physical processors that would be returned by the call NUMBER_OF_PROCESSORS(). It must also accept all declarations of scalar PROCESSOR arrangements. Other cases may be handled as well, depending on the implementation.

For compatibility with the Fortran 90 attribute syntax, an optional "::" may be inserted. The shape may also be specified with the DIMENSION attribute:

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

As in Fortran 90, an *explicit-shape-spec-list* in a *processors-decl* will override an explicit DIMENSION attribute:

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

Here RUBIKS_REVENGE is $4 \times 4 \times 4$ while RUBIK and SOMA are each $3 \times 3 \times 3$. (By the rules enunciated above, however, such a statement may not be completely portable because no HPF language processor is required to handle shapes of total sizes 27 and 64 simultaneously.)

Returning from a subprogram causes all processor arrangements declared local to that subprogram to become undefined. It is not HPF-conforming for any array or template to be distributed onto a processor arrangement at the time the processor arrangement becomes undefined unless at least one of two conditions holds:

- The array or template itself becomes undefined at the same time by virtue of returning from the subprogram.
- Whenever the subprogram is called, the processor arrangement is always locally defined in the same way, with identical lower bounds, and identical upper bounds.

> *Rationale.* Note that the second condition is slightly less stringent than requiring all expressions to be constant. This allows calls to NUMBER_OF_PROCESSORS or PROCESSORS_SHAPE to appear without violating the condition. (*End of rationale.*)

Variables in COMMON or having the SAVE attribute may be mapped to a locally declared processor arrangement, but because the first condition cannot hold for such variables (they don't become undefined), the second condition must be observed. This allows COMMON variables to work properly through the customary strategy of putting identical declarations in each scoping unit that needs to use them, while allowing the processor arrangements to which they may be mapped to depend on the value returned by NUMBER_OF_PROCESSORS.

> *Advice to implementors.* It may be desirable to have a way for the user to specify at compile time the number of physical processors on which the program is to be executed. This might be specified either by a implementation-dependent directive, for example, or through the programming environment (for example, as a UNIX command-line argument). Such facilities are beyond the scope of the HPF specification, but as food for thought we offer the following illustrative hypothetical examples:

```
!Declaration for multiprocessor by ABC Corporation
!ABC$ PHYSICAL PROCESSORS(8)
!Declaration for mpp by XYZ Incorporated
!XYZ$ PHYSICAL PROCESSORS(65536)
!Declaration for hypercube machine by PDQ Limited
!PDQ$ PHYSICAL PROCESSORS(2,2,2,2,2,2,2,2,2,2)
!Declaration for two-dimensional grid machine by TLA GmbH
!TLA$ PHYSICAL PROCESSORS(128,64)
!One of the preceding might affect the following
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS())
```

It may furthermore be desirable to have a way for the user to specify the precise mapping of the processor arrangement declared in a PROCESSORS statement to the physical processors of the executing hardware. Again, this might be specified either by a implementation-dependent directive or through the programming environment (for example, as a UNIX command-line argument); such facilities are beyond the scope of the HPF specification, but as food for thought we offer the following illustrative hypothetical example:

```
!PDQ$ PHYSICAL PROCESSORS(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))
```

This might specify that the first dimension of G should use hypercube axes 0, 1, 2 with a Gray-code ordering; the second dimension should use hypercube axes 6 through 11 with a Gray-code ordering; and the third dimension should use hypercube axes 3, 4, 5, and 12 with a binary ordering. (*End of advice to implementors.*)

3.8 TEMPLATE Directive

The TEMPLATE directive declares one or more templates, specifying for each the name, the rank (number of dimensions), and the extent in each dimension. It must appear in the *specification-part* of a scoping unit.

In the language of section 14.1.2 of the Fortran 90 standard, templates are local entities of class (1); therefore a template may not have the same name as a variable, named constant, internal procedure, etc., in the same scoping unit. Template names obey the rules for host and use association as other names in the list in section 12.1.2.2.1 of the Fortran 90 standard.

A template declared in a module has the default accessibility of the module.

Rationale. Because the name of a template is not a first-class entity in HPF, but must appear only in directives, it cannot appear in an *access-stmt* (PRIVATE or PUBLIC). If directives ever become full-fledged Fortran statements rather than structured comments, then it would be appropriate to allow the accessibility of a template to be controlled by listing its name in an *access-stmt*. (*End of rationale.*)

A template is simply an abstract space of indexed positions; it can be considered as an "array of nothings" (as compared to an "array of integers," say). A template may be used as an abstract *align-target* that may then be distributed.

H334 *template-directive* **is** `TEMPLATE` *template-decl-list*

H335 *template-decl* **is** *template-name* [(*explicit-shape-spec-list*)]

H336 *template-name* **is** *object-name*

Examples:

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

If the "`::`" syntax is used, then the declared templates may optionally be distributed in the same *combined-directive*. In this case all templates declared by the directive must have the same rank so that the `DISTRIBUTE` attribute will be meaningful. The `DIMENSION` attribute may also be used.

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

Templates are useful in the particular situation where one must align several arrays relative to one another but there is no need to declare a single array that spans the entire index space of interest. For example, one might want four $N \times N$ arrays aligned to the four corners of a template of size $(N+1) \times (N+1)$:

```
!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)
```

Templates may also be useful in making assertions about the mapping of dummy arguments (see Section 3.10).

Unlike arrays, templates cannot be in `COMMON`. So two templates declared in different scoping units will always be distinct, even if they are given the same name. The only way for two program units to refer to the same template is to declare the template in a module that is then used by the two program units.

Templates are not passed through the subprogram argument interface. The template to which a dummy argument is aligned is always distinct from the template to which the actual argument is aligned, though it may be a copy (see Section 3.9). On exit from a subprogram, an HPF implementation arranges that the actual argument is aligned with the same template with which it was aligned before the call.

Returning from a subprogram causes all templates declared local to that subprogram to become undefined. It is not HPF-conforming for any variable to be aligned to a template at the time the template becomes undefined unless at least one of two conditions holds:

- The variable itself becomes undefined at the same time by virtue of returning from the subprogram.

- Whenever the subprogram is called, the template is always locally defined in the same way, with identical lower bounds, identical upper bounds, and identical distribution information (if any) onto identically defined processor arrangements (see Section 3.7).

  Rationale. (Note that this second condition is slightly less stringent than requiring all expressions to be constant. This allows calls to `NUMBER_OF_PROCESSORS` or `PROCESSORS_SHAPE` to appear without violating the condition.) (*End of rationale.*)

Variables in `COMMON` or having the `SAVE` attribute may be mapped to a locally declared template, but because the first condition cannot hold for such variable (they don't become undefined), the second condition must be observed.

3.9 INHERIT Directive

The `INHERIT` directive specifies that a dummy argument should be aligned to a copy of the template of the corresponding actual argument in the same way that the actual argument is aligned.

H337 *inherit-directive* **is** `INHERIT` *dummy-argument-name-list*

The `INHERIT` directive causes the named subprogram dummy arguments to have the `INHERIT` attribute. Only dummy arguments may have the `INHERIT` attribute. An object must not have both the `INHERIT` attribute and the `ALIGN` attribute. The `INHERIT` directive may appear only in a *specification-part* of a scoping unit.

If a dummy argument has the `TARGET` attribute and no explicit mapping attributes, then the `INHERIT` attribute is implicitly assumed. (See section 3.10.)

The `INHERIT` attribute specifies that the template for a dummy argument should be inherited, by making a copy of the template of the actual argument. Moreover, the `INHERIT` attribute implies a default distribution of `DISTRIBUTE * ONTO *`.

Note that this default distribution is not part of Subset HPF; if a program uses `INHERIT`, it must override the default distribution with an explicit mapping directive in order to conform to Subset HPF.

See Section 3.10 for further exposition. If an explicit mapping directive appears for the dummy argument, thereby overriding the default distribution, then the actual argument must be a whole array or a regular array section; it may not be an expression of any other form.

If none of the attributes `INHERIT`, `ALIGN`, and `DISTRIBUTE` is specified explicitly for a dummy argument, then the template of the dummy argument has the same shape as the dummy itself and the dummy argument is aligned to its template by the identity mapping.

An `INHERIT` directive may be combined with other directives, with the attributes stated in any order, more or less consistent with Fortran 90 attribute syntax.

Consider the following example:

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

```
      SUBROUTINE PROBATE(BREAD)
      REAL BREAD(9)
!HPF$ INHERIT BREAD
```

The inherited template of BREAD has shape [100]; element BREAD(I) is aligned with element 5 + 2*I of the inherited template and, since BREAD does not appear in a prescriptive DISTRIBUTE directive, it has a BLOCK(10) distribution.

3.10 Alignment, Distribution, and Subprogram Interfaces

Mapping directives may be applied to dummy arguments in the same manner as for other variables; such directives may also appear in interface blocks. However, there are additional options that may be used only with dummy arguments: asterisks, indicating that a specification is descriptive rather than prescriptive, and the INHERIT attribute.

First, consider the rules for the caller. If there is an explicit interface for the called subprogram and that interface contains mapping directives (whether prescriptive or descriptive) for the dummy argument in question, the actual argument will be remapped if necessary to conform to the directives in the explicit interface. The template of the dummy will then be as declared in the interface. If there is no explicit interface, then actual arguments that are whole arrays or array sections not involving vector subscripts may be remapped at the discretion of the language processor; the values of other expressions may be mapped in any manner at the discretion of the language processor.

> *Rationale.* The caller is required to treat descriptive directives in an explicit interface as if they were prescriptive so that the directives in the interface may be an exact textual copy of the directives appearing in the subprogram. If the *caller* enforces descriptive directives as if they were prescriptive, then the descriptive directives in the *called* routine will in fact be correct descriptions. (*End of rationale.*)

In order to describe explicitly the distribution of a dummy argument, the template that is subject to distribution must be determined. A dummy argument always has a fresh template to which it is ultimately aligned; this template is constructed in one of three ways:

- If the dummy argument appears explicitly as an *alignee* in an ALIGN directive, its template is specified by the *align-target*.
- If the dummy argument is not explicitly aligned and does not have the INHERIT attribute, then the template has the same shape and bounds as the dummy argument; this is called the *natural template* for the dummy.
- If the dummy argument is not explicitly aligned and does have the INHERIT attribute, then the template is "inherited" from the actual argument according to the following rules:
  - If the actual argument is a whole array, the template of the dummy is a copy of the template with which the actual argument is ultimately aligned.
  - If the actual argument is an array section of array A where no subscript is a vector subscript, then the template of the dummy is a copy of the template with which A is ultimately aligned.

– If the actual argument is any other expression, the shape and distribution of the template may be chosen arbitrarily by the language processor (and therefore the programmer cannot know anything *a priori* about its distribution).

In all of these cases, we say that the dummy has an *inherited template* rather than a natural template.

Consider the following example:

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

The two array sections FRUG(1:40:3) and TWIST(1:40:3) are mapped onto abstract processors in the same manner:

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											

However, the subroutine TERPSICHORE will view them in different ways because it inherits the template for the second dummy but not the first:

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

Therefore the template of TANGO is a copy of the 128 element template of the whole array TWIST. The template is mapped like this:

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

`TANGO(I)` is aligned with element 3*I-2 of the template. But the template of `FOXTROT` has the same size 14 as `FOXTROT` itself. The actual argument, `FRUG(1:40:3)` is mapped to the 16 processors in this manner:

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

It would be reasonable to understand the mapping of the template of `FOXTROT` to coincide with the layout of the array section:

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											

but we shall see that this is not permitted in HPF. Within subroutine `TERPSICHORE` it would be correct to make the descriptive assertion

```
!HPF$ DISTRIBUTE TANGO *(BLOCK)
```

but it would not be correct to declare

```
!HPF$ DISTRIBUTE FOXTROT *(BLOCK)                  !Nonconforming
```

Each of these asserts that the template of the specified dummy argument is already distributed `BLOCK` on entry to the subroutine. The shape of the template for `TANGO` is `[128]`, inherited (copied) from the array `TWIST`, whose section was passed as the corresponding actual argument, and that template does indeed have a `BLOCK` distribution. But the shape of the template for `FOXTROT` is `[14]`; the layout of the elements of the actual argument `FRUG(1:40:3)` (3 on the first processor, 3 on the second processor, 2 on the third processor, 3 on the fourth processor, ...) cannot properly be described as a `BLOCK` distribution of a length-14 template, so the `DISTRIBUTE` declaration for `FOXTROT` shown above would indeed be erroneous.

On the other hand, the layout of `FRUG(1:40:3)` can be described in terms of an alignment to a length-128 template which can be described by an explicit `TEMPLATE` declaration (see Section 3.8), so the directives

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

could be correctly included in TERPSICHORE to describe the layout of FOXTROT on entry to the subroutine without using an inherited template.

The simplest case is the use of the INHERIT attribute alone. If a dummy argument has the INHERIT attribute and no explicit DISTRIBUTE attribute, the net effect is to tell the compiler to leave the data exactly where it is—and not attempt to remap the actual argument. The dummy argument will be mapped in exactly the same manner as the actual argument; the subprogram must be compiled in such a way as to work correctly no matter how the actual argument may be mapped onto abstract processors. (It has this effect because an INHERIT attribute on a dummy D implicitly specifies the default distribution

```
!HPF$ DISTRIBUTE D * ONTO *
```

rather than allowing the compiler to choose any distribution it pleases for the dummy argument. The meaning of this implied DISTRIBUTE directive is discussed below.)

In the general case of a DISTRIBUTE directive, where every *distributee* is a dummy argument, either the *dist-format-clause* or the *dist-target*, or both, may begin with, or consist of, an asterisk.

- Without an asterisk, a *dist-format-clause* or *dist-target* is prescriptive; the clause describes a distribution and constitutes a request of the language processor to make it so. This might entail remapping or copying the actual argument at run time in order to satisfy the requested distribution for the dummy.

- Starting with an asterisk, a *dist-format-clause* or *dist-target* is descriptive; the clause describes a distribution and constitutes an assertion to the language processor that it will already be so. The programmer claims that, for every call to the subprogram, the actual argument will be such that the stated distribution already describes the mapping of that data. (The intent is that if the argument is passed by reference, no movement of the data will be necessary at run time. All this is under the assumption that the language processor has observed all other directives. While a conforming HPF language processor is not required to obey mapping directives, it should handle descriptive directives with the understanding that their implied assertions are relative to this assumption.)

- Consisting of only an asterisk, a *dist-format-clause* or *dist-target* is transcriptive; the clause says nothing about the distribution but constitutes a request of the language processor to copy that aspect of the distribution from that of the actual argument. (The intent is that if the argument is passed by reference, no movement of the data will be necessary at run time.) Note that the transcriptive case, whether explicit or implicit, is not included in Subset HPF.

It is possible that, in a single DISTRIBUTE directive, the *dist-format-clause* might have an asterisk but not the *dist-target*, or vice versa.

These examples of DISTRIBUTE directives for dummy arguments illustrate the various combinations:

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

The language processor should do whatever it takes to cause URANIA to have a CYCLIC distribution on the processor arrangement GALILEO.

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

The language processor should do whatever it takes to cause POLYHYMNIA to be distributed onto the processor arrangement ELVIS, using whatever distribution format it currently has (which might be on some other processor arrangement). (You can't say this in Subset HPF.)

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

The language processor should do whatever it takes to cause THALIA to have a CYCLIC distribution on the processor arrangement FLIP; THALIA already has a cyclic distribution, though it might be on some other processor arrangement.

```
!HPF$ DISTRIBUTE CALLIOPE (CYCLIC) ONTO *HOMER
```

The language processor should do whatever it takes to cause CALLIOPE to have a CYCLIC distribution on the processor arrangement HOMER; CALLIOPE is already distributed onto HOMER, though it might be with some other distribution format.

```
!HPF$ DISTRIBUTE MELPOMENE * ONTO *EURIPIDES
```

MELPOMENE is asserted to already be distributed onto EURIPIDES; use whatever distribution format the actual argument had so, if possible, no data movement should occur. (You can't say this in Subset HPF.)

```
!HPF$ DISTRIBUTE CLIO *(CYCLIC) ONTO *HERODOTUS
```

CLIO is asserted to already be distributed CYCLIC onto HERODOTUS so, if possible, no data movement should occur.

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

The language processor should do whatever it takes to cause EUTERPE to have a CYCLIC distribution onto whatever processor arrangement the actual was distributed onto. (You can't say this in Subset HPF.)

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

The mapping of ERATO should not be changed from that of the actual argument. (You can't say this in Subset HPF.)

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

ARTHUR_MURRAY is asserted to already be distributed CYCLIC onto whatever processor arrangement the actual argument was distributed onto, and no data movement should occur. (You can't say this in Subset HPF.)

Please note that DISTRIBUTE ERATO * ONTO * does not mean the same thing as

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

This latter means: ERATO is asserted to already be distributed * (that is, on-processor) onto whatever processor arrangement the actual was distributed onto. Note that the processor arrangement is necessarily scalar in this case.

One may omit either the *dist-format-clause* or the *dist-onto-clause* for a dummy argument. If such a clause is omitted and the dummy argument has the INHERIT attribute, then the compiler must handle the directive as if * or ONTO * had been specified explicitly. If such a clause is omitted and the dummy does not have the INHERIT attribute, then the compiler may choose the distribution format or a target processor arrangement arbitrarily. Examples:

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

WHEEL_OF_FORTUNE is asserted to already be CYCLIC. As long as it is kept CYCLIC, it may be remapped it onto some other processor arrangement, but there is no reason to.

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

DAVID_LETTERMAN is asserted to already be distributed on TV in some fashion. The distribution format may be changed as long as DAVID_LETTERMAN is kept on TV. (Note that this declaration must be made in attributed form; the statement form

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

does not conform to the syntax for a DISTRIBUTE directive.)

The asterisk convention allows the programmer to make claims about the pre-existing distribution of a dummy based on knowledge of the mapping of the actual argument. But what claims may the programmer correctly make?

If the dummy argument has an inherited template, then the subprogram may contain directives corresponding to the directives describing the actual argument. Sometimes it is necessary, as an alternative, to introduce an explicit named template (using a TEMPLATE directive) rather than inheriting a template; an example of this (GURF) appears above, near the beginning of this section.

If the dummy argument has a natural template (no INHERIT attribute) then things are more complicated. In certain situations the programmer is justified in inferring a pre-existing distribution for the natural template from the distribution of the actual's template, that is, the template that would have been inherited if the INHERIT attribute had been specified. In all these situations, the actual argument must be a whole array or array section, and the template of the actual must be coextensive with the array along any axes having a distribution format other than "*."

If the actual argument is a whole array, then the pre-existing distribution of the natural template of the dummy is identical to that of the actual argument.

If the actual argument is an array section, then, from each *section-subscript* and the distribution format for the corresponding axis of the array being subscripted, one constructs an axis distribution format for the corresponding axis of the natural template:

- If the *section-subscript* is scalar and the array axis is collapsed (as by an ALIGN directive) then no entry should appear in the distribution for the natural template.

- If the *section-subscript* is a *subscript-triplet* and the array axis is collapsed (as by an ALIGN directive), then * should appear in the distribution for the natural template.

- If the *section-subscript* is scalar and the array axis corresponds to an actual template axis distributed `*`, then no entry should appear in the distribution for the natural template.

- If the *section-subscript* is a *subscript-triplet* and the array axis corresponds to an actual template axis distributed `*`, then `*` should appear in the distribution for the natural template.

- If the *section-subscript* is a *subscript-triplet* $l:u:s$ and the array axis corresponds to an actual template axis distributed BLOCK(n) (which might have been specified as simply BLOCK, but there will be some n that describes the resulting distribution) and LB is the lower bound for that axis of the array, then BLOCK(n/s) should appear in the distribution for the natural template, *provided* that s divides n evenly and that $l - LB < s$.

- If the *section-subscript* is a *subscript-triplet* $l:u:s$ and the array axis corresponds to an actual template axis distributed CYCLIC(n) (which might have been specified as simply CYCLIC, in which case $n = 1$) and LB is the lower bound for that axis of the array, then CYCLIC(n/s) should appear in the distribution for the natural template, *provided* that s divides n evenly and that $l - LB < s$.

If the situation of interest is not described by the cases listed above, no assertion about the distribution of the natural template of a dummy is HPF-conforming.

Here is a typical example of the use of this feature. The main program has a two-dimensional array TROGGS, which is to be processed by a subroutine one column at a time. (Perhaps processing the entire array at once would require prohibitive amounts of temporary space.) Each column is to be distributed across many processors.

```
      REAL TROGGS(1024,473)
!HPF$ DISTRIBUTE TROGGS(BLOCK,*)
      DO J=1,473
        CALL WILD_THING(TROGGS(:,J))
      END DO
```

Each column of TROGGS has a BLOCK distribution. The rules listed above justify the programmer in saying so:

```
      SUBROUTINE WILD_THING(GROOVY)
      REAL GROOVY(:)
!HPF$ DISTRIBUTE GROOVY *(BLOCK) ONTO *
```

Consider now the ALIGN directive. The presence or absence of an asterisk at the start of an *align-spec* has the same meaning as in a *dist-format-clause*: it specifies whether the ALIGN directive is descriptive or prescriptive, respectively.

If an *align-spec* that does not begin with `*` is applied to a dummy argument, the meaning is that the dummy argument will be forced to have the specified alignment on entry to the subprogram (which may require temporarily remapping the data of the actual argument or a copy thereof).

Note that a dummy argument may also be used as an *align-target*.

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

In this example the first dummy argument, TSAR, is allowed to remain aligned with the corresponding actual argument, while the second dummy argument, CZAR, is forced to be aligned with the first dummy argument. If the two actual arguments are already aligned, no remapping of the data will be required at run time; but the subprogram will operate correctly even if the actual arguments are not already aligned, at the cost of remapping the data for the second dummy argument at run time.

If the *align-spec* begins with "*", then the *alignee* must be a dummy argument and the directive must be ALIGN and not REALIGN. The "*" indicates that the ALIGN directive constitutes a guarantee on the part of the programmer that, on entry to the subprogram, the indicated alignment will already be satisfied by the dummy argument, without any action to remap it required at run time. For example:

```
      SUBROUTINE GRUNGE(PLUNGE,SPONGE)
      REAL PLUNGE(1000),SPONGE(1000)
!HPF$ INHERIT SPONGE
!HPF$ ALIGN PLUNGE WITH *SPONGE
```

This asserts that, for every J in the range 1:1000, on entry to subroutine GRUNGE, the directives in the program have specified that PLUNGE(J) is currently mapped to the same abstract processor as SPONGE(J). (The intent is that if the language processor has in fact honored the directives, then no interprocessor communication will be required to achieve the specified alignment.)

The alignment of a general expression is up to the language processor and therefore unpredictable by the programmer; but the alignment of whole arrays and array sections is predictable. In the code fragment

```
      REAL FIJI(5000),SQUEEGEE(2000)
!HPF$ ALIGN SQUEEGEE(K) WITH FIJI(2*K)
      CALL GRUNGE(FIJI(2002:4000:2),SQUEEGEE(1001:))
```

it is true that every element of the array section SQUEEGEE(1001:) is aligned with the corresponding element of the array section FIJI(2002:4000:2), so the claim made in subroutine GRUNGE is satisfied by this particular call.

Under certain circumstances, it may be possible to specify that one dummy argument be remapped if necessary and then to specify that another dummy will then be aligned with it:

```
      SUBROUTINE MURKY(THINK, DENSE)
!HPF$ PROCESSORS GUNK(32)
!HPF$ DISTRIBUTE (BLOCK) ONTO GUNK :: DENSE
!HPF$ ALIGN WITH *DENSE :: THICK
```

Note that the programmer cannot be justified in descriptively asserting that THICK will be aligned with DENSE after its remapping unless the remapping is fully specified (that is, no part of the remapping is left to the compiler to choose). Therefore an explicit processors arrangement necessarily appears in the example. The caller must ensure that the first actual argument is appropriately mapped onto an identical processors arrangement.

It is not permitted to say simply "ALIGN WITH *"; an *align-target* must follow the asterisk. (The proper way to say "accept any alignment" is INHERIT.)

If a dummy argument has no explicit ALIGN or DISTRIBUTE attribute, then the compiler provides an implicit alignment and distribution specification, one that could have been described explicitly without any "assertion asterisks".

The rules on the interaction of the REALIGN and REDISTRIBUTE directives with a subprogram argument interface are:

1. A dummy argument may be declared DYNAMIC. However, it is subject to the general restrictions concerning the use of the name of an array to stand for its associated template.

2. If an array or any section thereof is accessible by two or more paths, it is not HPF-conforming to remap it through any of those paths. For example, if an array is passed as an actual argument, it is forbidden to realign that array, or to redistribute an array or template to which it was aligned at the time of the call, until the subprogram has returned from the call. This prevents nasty aliasing problems. An example:

```
      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$ REDISTRIBUTE A             !nonconforming
      ...
      END
```

   Situations such as this are forbidden, for the same reasons that an assignment to A at the statement marked "nonconforming" would also be forbidden. In general, in *any* situation where assignment to a variable would be nonconforming by reason of aliasing, remapping of that variable by an explicit REALIGN or REDISTRIBUTE directive is also forbidden.

An overriding principle is that any mapping or remapping of arguments is not visible to the caller. This is true whether such remapping is implicit (in order to conform to prescriptive directives, which may themselves be explicit or implicit) or explicit (specified by REALIGN or REDISTRIBUTE directives). When the subprogram returns and the caller resumes execution, all objects accessible to the caller after the call are mapped exactly as they were before the call. It is not possible for a subprogram to change the mapping of any object in a manner visible to its caller, not even by means of REALIGN and REDISTRIBUTE.

> *Advice to implementors.* There are several implementation strategies for achieving this behavior. For example, one may be able to use a copy-in/copy-out strategy for arguments that require remapping on subprogram entry. Alternatively, one may be able to remap the actual argument on entry and remap again on exit to restore the original mapping. (*End of advice to implementors.*)

There is one sticky point in preserving this principle: a recent Fortran 90 interpretation states:

> If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument on invocation of the procedure.
>
> If the dummy argument has the TARGET attribute and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript:
>
> 1. Any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure.
> 2. When execution of the procedure completes, any pointers associated with the dummy argument remain associated with the actual argument.
>
> If the dummy argument has the TARGET attribute and the corresponding actual argument does not have the TARGET attribute or is an array section with a vector subscript, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

In order to support this behavior in the face of implicit remapping across the subprogram interface, HPF imposes the following restriction:

> If, on invocation of a procedure P: (a) a dummy argument has the TARGET attribute, and (b) the corresponding actual argument has the TARGET attribute and is not an array section with a vector subscript (and therefore is an object A or a section of an array A), then the program is not HPF-conforming unless:
>
> 1. No remapping of the actual argument occurs during the call; or
> 2. the remainder of program execution would be unaffected if
>   (a) each pointer associated with any portion of the dummy argument or with any portion of A during execution of P were to acquire undefined pointer association status on exit from P; and
>   (b) each pointer associated with any portion of A before the call were to acquire undefined pointer association status on entry to P and, if not reassigned during execution of P, were to be restored on exit to the pointer association status it had before entry.

Note that if a dummy argument has the TARGET attribute and no explicit mapping attributes, then the INHERIT attribute is implicitly assumed (see section 3.9); therefore no remapping occurs for such a dummy argument and there is no problem.

Section 4

Data Parallel Statements and Directives

The purpose of the FORALL statement and construct is to provide a convenient syntax for simultaneous assignments to large groups of array elements. Such assignments lie at the heart of the data parallel computations that HPF is designed to express. The multiple assignment functionality it provides is very similar to that provided by the array assignment statement and the WHERE construct in Fortran 90. FORALL differs from these constructs in its syntax, which is intended to be more suggestive of local operations on each element of an array, and in its generality, which allows a larger class of array sections to be specified. In addition, a FORALL may call user-defined functions on the elements of an array, simulating Fortran 90 elemental function invocation (albeit with a different syntax).

HPF defines a new procedure attribute, PURE, to declare the class of functions that may be invoked in this way. Both single-statement and block FORALL forms are defined in this Section, as well as the PURE attribute and constraints arising from the use of PURE.

HPF also defines a new directive, INDEPENDENT. The purpose of the INDEPENDENT directive is to allow the programmer to give additional information to the compiler. The user can assert that no data object is defined by one iteration of a DO loop and used (read or written) by another; similar information can be provided about the combinations of index values in a FORALL statement or construct. Such information is sometimes valuable to enable compiler optimizations, but may require knowledge of the application that is available only to the programmer. Therefore, HPF allows a user to specify these assertions, on which the compiler may in turn rely in its translation process. If the assertion is true, the semantics of the program are not changed; if it is false, the program is not HPF-conforming and has no defined meaning.

4.1 The FORALL Statement

Fortran 90 places several restrictions on array assignments. In particular, it requires that operands of the right side expressions be conformable with the left hand side array. These restrictions can be relaxed by introducing the element array assignment statement, usually referred to as the FORALL statement. This statement is used to specify an array assignment in terms of array elements or groups of array sections, possibly masked with a scalar logical expression. In functionality, it is similar to array assignment statements and WHERE statements. The FORALL statement essentially preserves the semantics of Fortran 90 array assignments and allows for convenient assignments like

```
FORALL ( i=1:n, j=1:m ) a(i,j)=i+j
```

as opposed to standard Fortran 90

```
a = SPREAD((/(i,i=1,n)/), DIM=2, NCOPIES=m) +    &
    SPREAD((/(i,i=1,m)/), DIM=1, NCOPIES=n)
```

It can also express more general array sections than the standard triplet notation for array expressions. For example,

```
FORALL ( i = 1:n ) a(i,i) = b(i)
```

assigns to the elements on the main diagonal of array `a`.

> *Rationale.* It is important to note, however, that `FORALL` is not intended to be a general parallel construct; for example, it does not express pipelined computations or MIMD computation well. This was an explicit design decision made in order to simplify the construct and promote agreement on the statement's semantics. (*End of rationale.*)

4.1.1 General Form of Element Array Assignment

Rule R216 in the Fortran 90 standard for *action-stmt* is extended to include the *forall-stmt*.

H401 *forall-stmt* **is** `FORALL` *forall-header forall-assignment*

H402 *forall-header* **is** (*forall-triplet-spec-list* [, *scalar-mask-expr*])

Constraint: Any procedure referenced in the *scalar-mask-expr* of a *forall-header* must be pure, as defined in Section 4.3.

> *Rationale.* Pure functions are guaranteed to be free of side effects. Therefore, they are safe to invoke in the *scalar-mask-expr*.
>
> Note that functions referenced in the *forall-triplet-spec-list* are not syntactically constrained as the *scalar-mask-expr* is. This is consistent with the handling of bounds expressions in `DO` loops. (*End of rationale.*)

H403 *forall-triplet-spec* **is** *index-name* = *subscript* : *subscript* [: *stride*]

Constraint: *index-name* must be a scalar integer variable.

Constraint: A *subscript* or *stride* in a *forall-triplet-spec-list* must not contain a reference to any *index-name* in the *forall-triplet-spec-list* in which it appears.

H404 *forall-assignment* **is** *assignment-stmt*
or *pointer-assignment-stmt*

Constraint: Any procedure referenced in a *forall-assignment*, including one referenced by a defined operation or assignment, must be pure as defined in Section 4.3.

Rationale. Pure functions are guaranteed to have no side effects, and thus have an unambiguous meaning when used in a `FORALL` statement. Experience also suggests that they form a useful class of functions for use in scientific computation, and are particularly useful when applied as data-parallel operations. For these reasons, there was a strong consensus to allow their use in `FORALL`. More general functions called from `FORALL` were also considered, but eventually rejected for lack of agreement on their desirability, ease of implementation, or the semantics of complex cases they allowed. (*End of rationale.*)

To determine the set of permitted values for each *index-name* in the *forall-header*, we introduce some simplifying notation. In the *forall-triplet-spec*, let

- $m1$ be first *subscript* ("lower bound");
- $m2$ be second *subscript* ("upper bound");
- $m3$ be the *stride*; and
- max be $\left\lfloor \frac{m2-m1+m3}{m3} \right\rfloor$.

If *stride* is missing, it is as if it were present with the value 1. *Stride* must not have the value 0. The set of permitted values is determined on entry to the statement and is $m1+(k-1)\times m3$, $k = 1, 2, ..., max$. If $max \leq 0$ for some *index-name*, the *forall-assignment* is not executed.

A `FORALL` statement assigns to memory locations specified by the *forall-assignment* for permitted values of the *index-name* variables. A program that causes multiple values to be assigned to the same location is not HPF-conforming and therefore has no defined meaning. This is a semantic constraint rather than a syntactic constraint, however; in general, it cannot be checked during compilation.

4.1.2 Interpretation of Element Array Assignments

Execution of an element array assignment consists of the following steps:

1. Evaluation in any order of the *subscript* and *stride* expressions in the *forall-triplet-spec-list*. The set of *valid combinations* of *index-name* values is then the Cartesian product of the sets defined by these triplets.

2. Evaluation of the *scalar-mask-expr* for all valid combinations of *index-name* values. The mask elements may be evaluated in any order. The set of *active combinations* of *index-name* values is the subset of the valid combinations for which the mask evaluates to `.TRUE.`.

3. Evaluation in any order of the *expr* and all expressions within *variable* (in the case of *assignment-stmt*) or *target* and all expressions within *pointer-object* (in the case of *pointer-assignment-stmt.*) of the *forall-assignment* for all active combinations of *index-name* values. In the case of pointer assignment where the *target* is not a pointer, the evaluation consists of identifying the object referenced rather than computing its value.

4. Assignment of the computed *expr* values to the corresponding *variable* locations (in the case of *assignment-stmt*) or the association of the *target* values with the corresponding *pointer-object* locations (in the case of *pointer-assignment-stmt*) for all active combinations of *index-name* values. The assignments or associations may be made in any order. In the case of a pointer assignment where the *target* is not a pointer, this assignment consists of associating the *pointer-object* with the object referenced.

If the scalar mask expression is omitted, it is as if it were present with the value `.TRUE.`

The scope of an *index-name* is the `FORALL` statement itself.

An *index-name* of a *forall-stmt* has statement scope, that is, its scope is the `FORALL` itself.

Rationale. This is the same as the treatment of a `DO` index in an implied-do list of a `DATA` statement. In both cases, the index is used only for its range of values; this was the basis for the similar treatment. (*End of rationale.*)

A *forall-stmt* is not HPF-conforming if the result of evaluating any expression in the *forall-header* affects or is affected by the evaluation of any other expression in the *forall-header*.

Rationale. This is consistent with the handling of `DO` loop bounds and strides. Disallowing references to impure functions in a *forall-triplet-spec-list* was suggested, but the analogy to `DO` bounds was considered too strong to overlook. Note that the *scalar-mask-expr* can only invoke pure functions, which are side-effect free. Therefore, the *scalar-mask-expr* cannot affect the values of the bounds. (*End of rationale.*)

A *forall-stmt* is not HPF-conforming if it causes any atomic data object to be assigned more than one value. A data object is atomic if it contains no subobjects. For the purposes of this restriction, any assignment (including array assignment or assignment to a variable of derived type) to a non-atomic object is considered to assign to all subobjects contained by that object.

Rationale. For example, an integer variable is an atomic object, but an array of integers is an object that is not atomic. Similarly, assignment to an array section is equivalent to assignments to each individual element (which may require further reductions when the array contains objects of derived type). This restriction allows cases such as

```
FORALL ( i = 1:10 ) a(indx(i)) = b(i)
```

if and only if `indx` contains no repeated values. Note that it restricts `FORALL` behavior, but not syntax. Syntactic restrictions to enforce this behavior would be either incomplete (ie. allow undefined behavior) or exclude conceptually legal programs.

Since a function called from a *forall-assignment* must be pure, it is impossible for that function's evaluation to affect other expressions' evaluations, either for the same combination of *index-name* values or for a different combination. In addition, it is possible that the compiler can perform more extensive optimizations because all functions are pure. (*End of rationale.*)

4.1.3 Examples of the FORALL Statement

```
FORALL (j=1:m, k=1:n) x(k,j) = y(j,k)
FORALL (k=1:n) x(k,1:m) = y(1:m,k)
```

These statements both copy columns 1 through n of array y into rows 1 through n of array x. This is equivalent to the standard Fortran 90 statement

```
x(1:n,1:m) = TRANSPOSE(y(1:m,1:n))
```

```
FORALL (i=1:n, j=1:n) x(i,j) = 1.0 / REAL(i+j-1)
```

This FORALL sets array element $x(i,j)$ to the value $\frac{1}{i+j-1}$ for values of i and j between 1 and n. In Fortran 90, the same operation can be performed by the statement

```
x(1:n,1:n) = 1.0/REAL( SPREAD((/(i,i=1,n)/),DIM=2,NCOPIES=n) &
    + SPREAD((/(j,j=1,n)/),DIM=1,NCOPIES=n) - 1 )
```

Note that the FORALL statement does not imply the creation of temporary arrays and is much more readable.

```
FORALL (i=1:n, j=1:n, y(i,j).NE.0.0) x(i,j) = 1.0 / y(i,j)
```

This statement takes the reciprocal of each nonzero element of array $y(1:n,1:n)$ and assigns it to the corresponding element of array x. Elements of y that are zero do not have their reciprocal taken, and no assignments are made to the corresponding elements of x. This is equivalent to the standard Fortran 90 statement

```
WHERE (y(1:n,1:n) .NE. 0.0) x(1:n,1:n) = 1 / y(1:n,1:n)
```

```
TYPE monarch
  INTEGER, POINTER :: p
END TYPE monarch
TYPE(monarch) :: a(n)
INTEGER, TARGET :: b(n)

! Set up a butterfly pattern
FORALL (j=1:n)  a(j)%p => b(1+IEOR(j-1,2**k))
```

This FORALL statement sets the elements of array a to point to a permutation of the elements of b. When $n = 8$ and $k = 1$, then elements 1 through 8 of a point to elements 3, 4, 1, 2, 7, 8, 5, and 6 of b, respectively. This requires a DO loop or other control flow in Fortran 90.

```
FORALL ( i=1:n ) x(indx(i)) = x(i)
```

This FORALL statement is equivalent to the Fortran 90 array assignment

```
x(indx(1:n)) = x(1:n)
```

If $indx$ contains a permutation of the integers from 1 to n, then the final contents of x will be a permutation of the original values. If $indx$ contains repeated values, neither the behavior of the FORALL nor the array assignment are defined by their respective standards.

```
FORALL (i=2:4) x(i) = x(i-1) + x(i) + x(i+1)
```

If this statement is executed with

$$x = [1.0, 20.0, 300.0, 4000.0, 50000.0]$$

then after execution the new values of array x will be

$$x = [1.0, 321.0, 4320.0, 54300.0, 50000.0]$$

This has the same effect as the Fortran 90 statement

```
x(2:4) = x(1:3) + x(2:4) + x(3:5)
```

Note that it does *not* have the same effect as the Fortran 90 loop

```
DO i = 2, 4
  x(i) = x(i-1) + x(i) + x(i+1)
END DO
```

```
FORALL (i=1:n) a(i,i) = x(i)
```

This FORALL statement sets the elements of the main diagonal of matrix a to the elements of vector x. This cannot be done by an array assignment in Fortran 90 unless EQUIVALENCE or WHERE is also used.

```
FORALL (i=1:4) a(i,ix(i)) = x(i)
```

This FORALL statement sets one element in each row of matrix a to an element of vector x. The particular elements in a are chosen by the integer vector ix. If

$$x = [10.0, 20.0, 30.0, 40.0]$$

$$ix = [1, 2, 2, 4]$$

and array a represents the matrix

$$\begin{array}{ccccc} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 2.0 & 2.0 & 2.0 \\ 3.0 & 3.0 & 3.0 & 3.0 & 3.0 \end{array}$$

before execution of the FORALL, then a will represent

$$\begin{array}{rrrrr} 10.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 20.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 30.0 & 2.0 & 2.0 & 2.0 \\ 3.0 & 3.0 & 3.0 & 3.0 & 40.0 \end{array}$$

after its execution. This operation cannot be accomplished with a single array assignment in Fortran 90.

```
FORALL (k=1:9) x(k) = SUM(x(1:10:k))
```

This FORALL statement computes nine sums of subarrays of x. (SUM is allowed in a FORALL because Fortran 90 intrinsic functions are pure; see Section 4.3.) If before the FORALL

$$x = [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0]$$

then after the FORALL

$$x = [55.0, 25.0, 22.0, 15.0, 7.0, 8.0, 9.0, 10.0, 11.0, 10.0]$$

This computation cannot be done by Fortran 90 array expressions alone.

4.1.4 Scalarization of the FORALL Statement

One way to understand the semantics of the FORALL statement is to exhibit a naive translation to scalar Fortran 90 code. We provide such a translation below.

> *Advice to implementors.* Note, however, that such a translation is meant for illustration rather than as the definitive reference to the FORALL semantics of or practical implementation in the compiler. In particular, implementing a FORALL using DO loops imposes an apparent order on the operations that is not implied by the formal definition. Additionally, compiler analysis of particular cases may allow significant simplification and optimization. For example, if the array assigned in a FORALL statement is not referenced in any other expression in the FORALL (including its use in functions called from the FORALL), it is legal and, on many machines, more efficient to perform the computations and final assignments in a single loop nest. Also note the discussion at the end of this section regarding other difficulties of a Fortran 90 translation. (*End of advice to implementors.*)

A *forall-stmt* of the form

FORALL (v_1=$l_1:u_1:s_1$, v_2=$l_1:u_2:s_2$, . . . , v_n=$l_n:u_n:s_n$, $mask$) a(e_1, . . . , e_m)=rhs

is equivalent to the following code:

```
! Evaluate subscript and stride expressions.
! These assignments may be executed in any order.
templ1 = l1
tempu1 = u1
temps1 = s1
templ2 = l2
tempu2 = u2
temps2 = s2
   ...
templn = ln
tempun = un
tempsn = sn
```

```
! Evaluate the scalar mask expression, and evaluate the
! forall-assignment subexpressions where the mask is true.
! The iterations of this loop nest may be executed in any order.
! The assignments in the loop body may be executed in any order,
! provided that the mask element is evaluated before any other
! expression in the same iteration.
! The loop body need not be executed atomically.
! The DO statements may be nested in any order
DO v1=templ1,tempu1,temps1
  DO v2=templ2,tempu2,temps2
    ...
      DO vn=templn,tempun,tempsn
        tempmask(v1,v2,...,vn) = mask
        IF (tempmask(v1,v2,...,vn)) THEN
          temprhs(v1,v2,...,vn) = rhs
          tempe1(v1,v2,...,vn) = e1
          tempe2(v1,v2,...,vn) = e2
          ....
          tempem(v1,v2,...,vn) = em
        END IF
      END DO
     ...
  END DO
END DO

! Perform the assignment of these values to the corresponding
! elements of the array on the left-hand side.
! The iterations of this loop nest may be executed in any order.
! The DO statements may be nested in any order.
DO v1=templ1,tempu1,temps1
  DO v2=templ2,tempu2,temps2
    ...
      DO vn=templn,tempun,tempsn
        IF (tempmask(v1,v2,...,vn)) THEN
          a(tempe1(v1,v2,...,vn),...,tempem(v1,v2,...,vn)) =     &
            temprhs(v1,v2,...,vn)
        END IF
      END DO
    ...
  END DO
END DO
```

The scalarization of a `FORALL` statement containing a pointer assignment is similar, replacing the assignments to *temprhs* and *a* with pointer assignments.

Advice to implementors. Several subtleties are not specified in the above outline to promote readability. When *rhs* is an array-valued expression, then several of the statements cannot be translated directly into Fortran 90. In particular, at least one of the e_i will be a triplet; both bounds and stride must be saved in $tempe_i$, possibly by using derived type assignment or adding a dimension to the data structure. The translation of the subscripts in the final assignment to a must also be generalized to handle triplets. Storage allocation for *temprhs* may be complicated by the fact that it must store arrays (possibly with different sizes for different values of $v_1, \ldots, v_n$). If the *forall-assignment* is a *pointer-assignment-stmt*, then a suitable derived type must be produced for *temprhs*. The assignments to $tempe_1, \ldots, tempe_m$ must, however, remain true (integer) assignments. Finally, there may also be more than seven indexes; this may forbid a direct translation on implementations that support a limited number of dimensions in arrays. (*End of advice to implementors.*)

4.1.5 Consequences of the Definition of the FORALL Statement

Rationale. The *scalar-mask-expr* may depend on the *index-name* values. This allows a wide range of masking operations.

A syntactic consequence of the semantic rule that no two execution instances of the body may assign to the same atomic data object is that each of the *index-name* variables must appear on the left-hand side of a *forall-assignment*. The converse is not true (i.e., using all *index-name* variables on the left-hand side does not guarantee there will be no interference). Because the condition is not sufficient, it does not appear a syntax constraint. This also allows for easier future extensions for private variables or other syntactic sugar.

Right-hand sides and expressions on the left hand side of a *forall-assignment* are defined as evaluated only for combinations of *index-names* for which the *scalar-mask-expr* evaluates to `.TRUE.`. This has implications when the masked computation might create an error condition. For example,

```
FORALL (i=1:n, y(i).NE.0.0) x(i) = 1.0 / y(i)
```

does not cause a division by zero. (*End of rationale.*)

4.2 The FORALL Construct

The `FORALL` construct is a generalization of the `FORALL` statement allowing multiple assignments, masked array assignments, and nested `FORALL` statements and constructs to be controlled by a single *forall-triplet-spec-list*.

4.2.1 General Form of the FORALL Construct

Rule R215 of the Fortran 90 standard for *executable-construct* is extended to include the *forall-construct*.

H405	*forall-construct*	**is**	`FORALL` *forall-header*
			forall-body-stmt
			[*forall-body-stmt*] ...
			`END FORALL`

H406 *forall-body-stmt*	**is**	*forall-assignment*
	or	*where-stmt*
	or	*where-construct*
	or	*forall-stmt*
	or	*forall-construct*

Constraint: Any procedure referenced in a *forall-body-stmt*, including one referenced by a defined operation or assignment, must be pure as defined in Section 4.3.

Constraint: If a *forall-stmt* or *forall-construct* is nested in a *forall-construct*, then the inner `FORALL` may not redefine any *index-name* used in the outer *forall-construct*.

Rationale. These statements are allowed in a `FORALL` construct because they are defined as forms of assignment in Fortran 90 and HPF. The intent is that *forall-construct*, like *forall-stmt*, is a block assignment rather than a general-purpose "parallel loop." (*End of rationale.*)

To determine the set of permitted values for an *index-name*, we introduce some simplifying notation. In the *forall-triplet-spec*, let

- $m1$ be the first *subscript* ("lower bound");
- $m2$ be the second *subscript* ("upper bound");
- $m3$ be the *stride*; and
- max be $\left\lfloor \frac{m2-m1+m3}{m3} \right\rfloor$.

If *stride* is missing, it is as if it were present with the value 1. The set of permitted values is determined on entry to the construct and is $m1 + (k-1) \times m3$, $k = 1, 2, ..., max$. The expression *stride* must not have the value 0. If for some *index-name* $max \leq 0$, no *forall-body-stmt* is executed.

Each assignment nested within a `FORALL` construct assigns to memory locations specified by the *forall-assignment* for permitted values of the *index-name* variables. A program that causes multiple values to be assigned to the same location by a single statement is not HPF-conforming and therefore has no defined meaning. An HPF-conforming program may, however, assign to the same location in syntactically different assignment statements. This is a semantic constraint rather than a syntactic constraint, however; in general, it cannot be checked during compilation.

4.2.2 Interpretation of the FORALL Construct

Execution of a `FORALL` construct consists of the following steps:

1. Evaluation in any order of the *subscript* and *stride* expressions in the *forall-triplet-spec-list*. The set of *valid combinations* of *index-name* values is then the Cartesian product of the sets defined by these triplets.

2. Evaluation of the *scalar-mask-expr* for all valid combinations of *index-name* values. The mask elements may be evaluated in any order. The set of *active combinations* of *index-name* values is the subset of the valid combinations for which the mask evaluates to `.TRUE.`

3. Execute the *forall-body-stmts* in the order they appear. Each statement is executed completely (that is, for all active combinations of *index-name* values) according to the following interpretation:

   (a) Statements in the *forall-assignment* category (i.e. assignment statements and pointer assignment statements) evaluate the *expr* and all expressions within *variable* (in the case of *assignment-stmt*) or *target* and all expressions within *pointer-object* (in the case of *pointer-assignment-stmt*) of the *forall-assignment* for all active combinations of *index-name* values. These evaluations may be done in any order. The *expr* values are then assigned to the corresponding *variable* locations (in the case of *assignment-stmt*) or the *target* values are associated with the corresponding *pointer-object* locations (in the case of *pointer-assignment-stmt*). The assignment or association operations may also be performed in any order.

   (b) Statements in the *where-stmt* and *where-construct* categories evaluate their *mask-expr* for all active combinations of values of *index-names*. All elements of all masks may be evaluated in any order. The `WHERE` statement's assignment (or assignments within the `WHERE` branch of the construct) are then executed in order using the above interpretation of array assignments within the `FORALL`, but the only array elements assigned are those selected by both the active *index-name* values and the `WHERE` mask. Finally, the assignments in the `ELSEWHERE` branch are executed if that branch is present. The assignments here are also treated as array assignments, but elements are only assigned if they are selected by both the active combinations and by the negation of the `WHERE` mask.

   (c) Statements in the *forall-stmt* and *forall-construct* categories first evaluate the *subscript* and *stride* expressions in the *forall-triplet-spec-list* for all active combinations of the outer `FORALL` constructs. The set of valid combinations of *index-names* for the inner `FORALL` is then the union of the sets defined by these bounds and strides for each active combination of the outer *index-names*, the outer *index names* being included in the combinations generated for the inner `FORALL`. The scalar mask expression is then evaluated for all valid combinations of the inner `FORALL`'s *index-names* to produce the set of active combinations. If there is no scalar mask expression, it is as if it were present with the constant value `.TRUE.`. Each statement in the inner `FORALL` is then executed for each active combination (of the inner `FORALL`), recursively following the interpretations given in this section.

If the scalar mask expression is omitted, it is as if it were present with the value `.TRUE.`.

The scope of an *index-name* is the `FORALL` construct itself. That is, the *index-name* defines a new variable that is only valid in the statements of the `FORALL` body. The same name may be used outside the `FORALL` construct as a local or global entity without conflict, and refers to a different entity when so used.

Rationale. This extends the Fortran 90 concept of "statement scope" to include entire constructs. The reasons for limiting the scope of the index are the same as for `FORALL` statement indices. However, tradtional statement scope is insufficient for a multi-statement construct; we therefore made the natural extension. (*End of rationale.*)

Each *forall-assignment* must obey the same restrictions in a *forall-construct* as in a simple *forall-stmt*. In addition, each *where-stmt* or assignment nested within a *where-construct* must obey these restrictions. (Note that any innermost statement within nested FORALL constructs must fall into one of these two categories.) For example, an assignment may not cause the same array element to be assigned more than once. Different statements may, however, assign to the same array element, and assignments made in one statement may affect the execution of a later statement.

4.2.3 Examples of the FORALL Construct

```
FORALL ( i=2:n-1, j=2:n-1 )
  a(i,j) = a(i,j-1) + a(i,j+1) + a(i-1,j) + a(i+1,j)
  b(i,j) = a(i,j)
END FORALL
```

This FORALL is equivalent to the two Fortran 90 statements

```
a(2:n-1,2:n-1) = a(2:n-1,1:n-2)+a(2:n-1,3:n)       &
                           +a(1:n-2,2:n-1)+a(3:n,2:n-1)
b(2:n-1,2:n-1) = a(2:n-1,2:n-1)
```

In particular, note that the assignment to array b uses the values of array a computed in the first statement, not the values before the FORALL began execution.

```
FORALL ( i=1:n-1 )
  FORALL ( j=i+1:n )
    a(i,j) = a(j,i)
  END FORALL
END FORALL
```

This FORALL construct assigns the transpose of the lower triangle of array a (i.e., the section below the main diagonal) to the upper triangle of a. For example, if $n = 5$ and a originally contained the matrix

$$\begin{array}{rrrrr} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 4.0 & 8.0 & 16.0 & 32.0 \\ 3.0 & 9.0 & 27.0 & 81.0 & 243.0 \\ 4.0 & 16.0 & 64.0 & 256.0 & 1024.0 \end{array}$$

then after the FORALL it would contain

$$\begin{array}{rrrrr} 0.0 & 1.0 & 2.0 & 3.0 & 4.0 \\ 1.0 & 1.0 & 4.0 & 9.0 & 16.0 \\ 2.0 & 4.0 & 8.0 & 27.0 & 64.0 \\ 3.0 & 9.0 & 27.0 & 81.0 & 256.0 \\ 4.0 & 16.0 & 64.0 & 256.0 & 1024.0 \end{array}$$

This cannot be done using array expressions without introducing mask expressions.

```
FORALL ( i=1:5 )
  WHERE ( a(i,:) .NE. 0.0 )
    a(i,:) = a(i-1,:) + a(i+1,:)
  ELSEWHERE
    b(i,:) = a(6-i,:)
  END WHERE
END FORALL
```

This `FORALL` construct, when executed with the input arrays

$$a = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 1.0 & 0.0 & 1.0 \\ 2.0 & 2.0 & 0.0 & 2.0 & 2.0 \\ 3.0 & 0.0 & 3.0 & 3.0 & 3.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}, \quad b = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 10.0 & 10.0 & 10.0 & 10.0 & 10.0 \\ 20.0 & 20.0 & 20.0 & 20.0 & 20.0 \\ 30.0 & 30.0 & 30.0 & 30.0 & 30.0 \\ 40.0 & 40.0 & 40.0 & 40.0 & 40.0 \end{pmatrix}$$

will produce as results

$$a = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 2.0 & 2.0 & 0.0 & 0.0 & 2.0 \\ 4.0 & 1.0 & 0.0 & 3.0 & 4.0 \\ 2.0 & 0.0 & 0.0 & 2.0 & 2.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}, \quad b = \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 10.0 & 10.0 & 10.0 & 2.0 & 10.0 \\ 20.0 & 20.0 & 0.0 & 20.0 & 20.0 \\ 30.0 & 2.0 & 30.0 & 30.0 & 30.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

Note that, as with `WHERE` statements in ordinary Fortran 90, assignments in the `WHERE` branch may affect computations in the `ELSEWHERE` branch.

4.2.4 Scalarization of the FORALL Construct

Advice to implementors. As with the `FORALL` statement, the following translations of `FORALL` constructs to `DO` loops are meant to illustrate the meaning, not necessarily to serve as an implementation guide. The caveats for the `FORALL` statement scalarization apply here as well. (*End of advice to implementors.*)

A *forall-construct* of the form:

```
FORALL (... e1 ... e2 ... en ...)
    s1
    s2
     ...
    sn
END FORALL
```

where each s_i is a *forall-assignment*, is equivalent to the following code:

```
temp1 = e1
temp2 = e2
 ...
tempn = en
FORALL (... temp1 ... temp2 ... tempn ...) s1
FORALL (... temp1 ... temp2 ... tempn ...) s2
   ...
FORALL (... temp1 ... temp2 ... tempn ...) sn
```

When the s_i are FORALL or WHERE statements or constructs, then the FORALL statements above must be replaced with FORALL constructs (since FORALL statements can only contain assignments). The scalarizations below must then be applied to the shortened FORALL constructs.

A *forall-construct* of the form:

```
FORALL ( v1=l1:u1:s1, mask1 )
  WHERE ( mask2 )
    a(l2:u2:s2) = rhs1
  ELSEWHERE
    a(l3:u3:s3) = rhs2
  END WHERE
END FORALL
```

is equivalent to the following code:

```
! Evaluate subscript and stride expressions.
! These assignments can be made in any order.
templ1 = l1
tempu1 = u1
temps1 = s1

! Evaluate the FORALL mask expression.
! The iterations of this loop may be executed in any order.
DO v1=templ1,tempu1,temps1
 tempmask1(v1) = mask1
END DO

! Evaluate the bounds and masks for the WHERE.
! The iterations of this loop may be executed in any order.
! The loop body need not be executed atomically.
DO v1=templ1,tempu1,temps1
  IF (tempmask1(v1)) THEN
    tempmask2(v1) = mask2
  END IF
END DO

! Evaluate the WHERE branch.
! The iterations of this loop may be executed in any order.
! The assignments in the loop body may be executed in any order.
! The loop body need not be executed atomically.
DO v1=templ1,tempu1,temps1
  IF (tempmask1(v1)) THEN
    tmpl2(v1) = l2
    tmpu2(v1) = u2
    tmps2(v1) = s2
    WHERE ( tempmask2(v1) )
      temprhs1(v1) = rhs1
    END WHERE
```

```
    END IF
  END DO
  ! The iterations of this loop may be executed in any order.
  ! The loop body need not be executed atomically.
  DO v1=templ1,tempu1,temps1
    IF (tempmask1(v1)) THEN
      WHERE ( tempmask2(v1) )
        a(tmpl2(v1):tmpu2(v1):tmps2(v1)) = temprhs1(v1)
      END WHERE
    END IF
  END DO

  ! Evaluate the ELSEWHERE branch.
  ! The iterations of this loop may be executed in any order.
  ! The assignments in the loop body may be executed in any order.
  ! The loop body need not be executed atomically.
  DO v1=templ1,tempu1,temps1
    IF (tempmask1(v1)) THEN
      tmpl3(v1) = l3
      tmpu3(v1) = u3
      tmps3(v1) = s3
      WHERE ( .NOT. tempmask2(v1) )
        temprhs2(v1) = rhs2
      END WHERE
    END IF
  END DO
  ! The iterations of this loop may be executed in any order.
  ! The loop body need not be executed atomically.
  DO v1=templ1,tempu1,temps1
    IF (tempmask1(v1)) THEN
      WHERE ( .NOT. tempmask2(v1) )
        a(tmpl3(v1):tmpu3(v1):tmps3(v1)) = temprhs2(v1)
      END WHERE
    END IF
  END DO
```

Advice to implementors. Note that the assignments to $tempmask_2$ and $temprhs_i$ are array assignments and require special treatment (including saving of shape information) similar to that for array assignments in the `FORALL` statement scalarization. The extension to multiple dimensions (in either the `FORALL` index space or the array dimensions) is straightforward. If there are multiple statements in a branch of the `WHERE` construct, each statement will generate two loops similar to those shown above. (*End of advice to implementors.*)

A *forall-construct* of the form:

```
FORALL ( v1=l1:u1:s1, mask1 )
  FORALL ( v2=l2:u2:s2, mask2 )
    a(e1) = rhs1
```

```
    b(e_2) = rhs_2
  END FORALL
END FORALL
```

is equivalent to the following Fortran 90 code:

```
! Evaluate subscript and stride expressions and outer mask.
! These assignments may be executed in any order.
templ_1 = l_1
tempu_1 = u_1
temps_1 = s_1
! The iterations of this loop may be executed in any order.
DO v_1=templ_1,tempu_1,temps_1
 tempmask_1(v_1) = mask_1
END DO

! Evaluate the inner FORALL bounds, etc
! The iterations of this loop may be executed in any order.
! The assignments in the loop body may be executed in any order,
! provided that the mask bounds are computed before the mask itself.
! The loop body need not be executed atomically.
DO v_1=templ_1,tempu_1,temps_1
  IF (tempmask_1(v_1)) THEN
    templ_2(v_1) = l_2
    tempu_2(v_1) = u_2
    temps_2(v_1) = s_2
    DO v_2 = templ_2(v_1),tempu_2(v_1),temps_2(v_1)
      tempmask_2(v_1,v_2) = mask_2
    END DO
  END IF
END DO

! Evaluate first statement
! The iterations of this loop may be executed in any order.
! The assignments in this loop body may be executed in any order.
! The loop body need not be executed atomically.
DO v_1=templ_1,tempu_1,temps_1
  IF (tempmask_1(v_1)) THEN
    DO v_2 = templ_2(v_1),tempu_2(v_1),temps_2(v_1)
      IF ( tempmask_2(v_1,v_2) ) THEN
        temprhs_1(v_1,v_2) = rhs_1
        tmpe_1(v_1,v_2) = e_1
      END IF
    END DO
  END IF
END DO
! The iterations of this loop may be executed in any order.
DO v_1=templ_1,tempu_1,temps_1
  IF (tempmask_1(v_1)) THEN
```

```
      DO v2 = templ2(v1),tempu2(v1),temps2(v1)
        IF ( tempmask2(v1,v2) ) THEN
          a(tmpe1(v1,v2)) = temprhs1(v1,v2)
        END IF
      END DO
    END IF
  END DO

  ! Evaluate second statement.
  ! Ordering constraints are as for the first statement.
  DO v1=templ1,tempu1,temps1
    IF (tempmask1(v1)) THEN
      DO v2 = templ2(v1),tempu2(v1),temps2(v1)
        IF ( tempmask2(v1,v2) ) THEN
          temprhs2(v1,v2) = rhs2
          tmpe2(v1,v2) = e2
        END IF
      END DO
    END IF
  END DO
  DO v1=templ1,tempu1,temps1
    IF (tempmask1(v1)) THEN
      DO v2 = templ2(v1),tempu2(v1),temps2(v1)
        IF ( tempmask2(v1,v2) ) THEN
          b(tmpe2(v1,v2)) = temprhs2(v1,v2)
        END IF
      END DO
    END IF
  END DO
```

Again, the extensions to higher dimensions are straightforward, as is the extension to deeper nesting levels.

Advice to implementors. Note that each statement at the deepest nesting level will generate two loops of the types shown. (*End of advice to implementors.*)

4.2.5 Consequences of the Definition of the FORALL Construct

Rationale.

A block `FORALL` means roughly the same thing as does replicating the `FORALL` header in front of each array assignment statement in the block, except that any expressions in the `FORALL` header are evaluated only once, rather than being re-evaluated before each of the statements in the body. The exceptions to this rule are nested `FORALL` statements and `WHERE` statements, which introduce syntactic and functional complications into the copying.

One may think of a block `FORALL` as synchronizing twice per contained assignment statement: once after handling the right-hand side and other expressions but before performing assignments, and once after all assignments have been performed but before commencing the next statement. In practice, appropriate analysis will often permit the compiler to eliminate unnecessary synchronizations.

In general, any expression in a `FORALL` is evaluated only for valid combinations of all surrounding *index-names* for which all the scalar mask expressions are `.TRUE.`.

Nested `FORALL` bounds and strides can depend on outer `FORALL` *index-names*. They cannot redefine those names, even temporarily (if they did, there would be no way to avoid multiple assignments to the same array element).

Statements can use the results of computations in lexically earlier statements, including computations done for other name values. However, an assignment never uses a value assigned in the same statement by another *index-name* value combination.

(*End of rationale.*)

4.3 Pure Procedures

A *pure function* is one that obeys certain syntactic constraints that ensure it produces no side effects. This means that the only effect of a pure function reference on the state of a program is to return a result—it does not modify the values, pointer associations, or data mapping of any of its arguments or global data, and performs no external I/O. A *pure subroutine* is one that produces no side effects except for modifying the values and/or pointer associations of `INTENT(OUT)` and `INTENT(INOUT)` arguments. These properties are declared by a new attribute (the `PURE` attribute) of the the procedure.

A pure procedure (i.e., function or subroutine) may be used in any way that a normal procedure can. However, a procedure is required to be pure if it is used in any of the following contexts:

- The mask or body of a `FORALL` statement or construct;
- Within the body of a pure procedure; or
- As an actual argument in a pure procedure reference.

Rationale.

The freedom from side effects of a pure function allows the function to be invoked concurrently in a `FORALL` without such undesirable consequences as nondeterminism, and additionally assists the efficient implementation of concurrent execution. Syntactic constraints (rather than semantic constraints on behavior) are used to enable compiler checking.

The HPF Journal of Development also proposes allowing elemental invocation of pure procedures with scalar arguments.

(*End of rationale.*)

4.3.1 Pure Procedure Declaration and Interface

If a user-defined procedure is used in a context that requires it to be pure, then its interface must be explicit in the scope of that use, and that interface must specify the `PURE` attribute. This attribute is specified in the *function-stmt* or *subroutine-stmt* by an extension of rules R1217 (for *prefix*) and R1220 (for *subroutine-stmt*) in the Fortran 90 standard. Rule R1216 (for *function-stmt*) is not changed, but is rewritten here as Rule H409 for clarity.

H407 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

H408 *prefix-spec* **is** *type-spec*
or RECURSIVE
or PURE
or *extrinsic-prefix*

H409 *function-stmt* **is** [*prefix*] FUNCTION *function-name function-stuff*

H410 *function-stuff* **is** ([*dummy-arg-name-list*]) [RESULT (*result-name*)]

H411 *subroutine-stmt* **is** [*prefix*] SUBROUTINE *subroutine-name subroutine-stuff*

H412 *subroutine-stuff* **is** [([*dummy-arg-list*])]

Constraint: A *prefix* must contain at most one of each variety of *prefix-spec*.

Constraint: The *prefix* of a *subroutine-stmt* must not contain a *type-spec*.

(For a discussion of the *extrinsic-prefix* (Rule H601), see Section 6.2.)

Intrinsic functions, including the HPF intrinsic functions, are always pure and require no explicit declaration of this fact. Intrinsic subroutines are pure if they are elemental (i.e., MVBITS) but not otherwise. Functions and subroutines in the HPF library are declared to be pure. A statement function is pure if and only if all functions that it references are pure.

A procedure with the PURE attribute is referred to as a "pure procedure" in the following constraints.

4.3.1.1 Pure function definition

The following constraints are added to Rule R1215 in Section 12.5.2.2 of the Fortran 90 standard (defining *function-subprogram*):

Constraint: The *specification-part* of a pure function must specify that all dummy arguments have INTENT(IN) except procedure arguments and arguments with the POINTER attribute.

Constraint: A local variable declared in the *specification-part* or *internal-subprogram-part* of a pure function must not have the SAVE attribute.

> *Advice to users.* Note local variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the SAVE attribute; therefore, such initialization is also disallowed. (*End of advice to users.*)

Constraint: The *execution-part* and *internal-subprogram-part* of a pure function may not use a dummy argument, a global variable, or an object that is storage associated with a global variable, or a subobject thereof, in the following contexts:

- As the assignment variable of an *assignment-stmt*;
- As a DO variable or implied DO variable, or as an *index-name* in a *forall-triplet-spec*;
- As an *input-item* in a *read-stmt*;
- As an *internal-file-unit* in a *write-stmt*;

- As an `IOSTAT=` or `SIZE=` specifier in an I/O statement.
- In an *assign-stmt*;
- As the *pointer-object* or *target* of a *pointer-assignment-stmt*;
- As the *expr* of an *assignment-stmt* whose assignment variable is of a derived type, or is a pointer to a derived type, that has a pointer component at any level of component selection;
- As an *allocate-object* or *stat-variable* in an *allocate-stmt* or *deallocate-stmt*, or as a *pointer-object* in a *nullify-stmt*; or
- As an actual argument associated with a dummy argument with `INTENT (OUT)` or `INTENT(INOUT)` or with the `POINTER` attribute.

Constraint: Any procedure referenced in a pure function, including one referenced via a defined operation or assignment, must be pure.

Constraint: A dummy argument or the dummy result of a pure function may be explicitly aligned only with another dummy argument or the dummy result, and may not be explicitly distributed or given the `INHERIT` attribute.

Constraint: In a pure function, a local variable may be explicitly aligned only with another local variable, a dummy argument, or the result variable. A local variable may not be explicitly distributed.

Constraint: In a pure function, a dummy argument, local variable, or the result variable must not have the `DYNAMIC` attribute.

Constraint: In a pure function, a global variable must not appear in a *realign-directive* or *redistribute-directive*.

Constraint: A pure function must not contain a *backspace-stmt*, *close-stmt*, *endfile-stmt*, *inquire-stmt*, *open-stmt*, *print-stmt*, *rewind-stmt*, or a *read-stmt* or *write-stmt* whose *io-unit* is an *external-file-unit* or `*`.

Constraint: A pure function must not contain a *pause-stmt* or *stop-stmt*.

The above constraints are designed to guarantee that a pure function is free from side effects (i.e., modifications of data visible outside the function), which means that it is safe to reference concurrently, as explained earlier.

Rationale.

It is worth mentioning why the above constraints are sufficient to eliminate side effects.

The first constraint (requiring explicit `INTENT(IN)`) declares behavior that is ensured by the following rules. It is not technically necessary, but is included for consistency with the explicit declaration rules for defined operators. Note that `POINTER` arguments may not have the `INTENT` attribute; the restrictions below ensure that `POINTER` arguments also behave as if they had `INTENT(IN)`, for both the argument itself and the object pointed to.

The second constraint (disallowing `SAVE` variables) ensures that a pure function does not retain an internal state between calls, which would allow side-effects between calls to the same procedure.

The third constraint (the restrictions on use of global variables and dummy arguments) ensures that dummy arguments and global variables are not modified by the function. In the case of a dummy or global pointer, this applies to both its pointer association and its target value, so it cannot be subject to a pointer assignment or to an `ALLOCATE`, `DEALLOCATE`, or `NULLIFY` statement. Incidentally, these constraints imply that only local variables and the dummy result variable can be subject to assignment or pointer assignment.

In addition, a dummy or global data object cannot be the *target* of a pointer assignment (i.e., it cannot be used as the right hand side of a pointer assignment to a local pointer or to the result variable), for then its value could be modified via the pointer. (An alternative approach would be to allow such objects to be pointer targets, but disallow assignments to those pointers; syntactic constraints to allow this would be even more draconian than these.)

In connection with the last point, it should be noted that an ordinary (as opposed to pointer) assignment to a variable of derived type that has a pointer component at any level of component selection may result in a *pointer* assignment to the pointer component of the variable. That is certainly the case for an intrinsic assignment. In that case, the expression on the right hand side of the assignment has the same type as the assignment variable, and the assignment results in a pointer assignment of the pointer components of the expression result to the corresponding components of the variable (see section 7.5.1.5 of the Fortran 90 standard). However, it may also be the case for a *defined* assignment to such a variable, even if the data type of the expression has no pointer components; the defined assignment may still involve pointer assignment of part or all of the expression result to the pointer components of the assignment variable. Therefore, a dummy or global object cannot be used as the right hand side of any assignment to a variable of derived type with pointer components, for then it, or part of it, might be the target of a pointer assignment, in violation of the restriction mentioned above.

(Incidentally, the last two paragraphs only prevent the reference of a dummy or global object as the *only* object on the right hand side of a pointer assignment or an assignment to a variable with pointer components. There are no constraints on its reference as an operand, actual argument, subscript expression, etc. in these circumstances.)

Finally, a dummy or global data object cannot be used in a procedure reference as an actual argument associated with a dummy argument of `INTENT(OUT)` or `INTENT (INOUT)` or with a dummy pointer, for then it may be modified by the procedure reference. This constraint, like the others, can be statically checked, since any procedure referenced within a pure function must be either a pure function, which does not modify its arguments, or a pure subroutine, whose interface must specify the `INTENT` or `POINTER` attributes of its arguments (see below). Incidentally, notice that in this context it is assumed that an actual argument associated with a dummy pointer is modified, since Fortran 90 does not allow its intent to be specified.

The fourth constraint (only pure procedures may be called) ensures that all procedures called from a pure function are themselves side-effect free, except, in the case of subroutines, for modifying actual arguments associated with dummy pointers or dummy arguments with `INTENT(OUT)` or `INTENT(INOUT)`. As we have just explained, it can be checked that global or dummy objects are not used in such arguments, which would violate the required side-effect freedom.

Constraints 5 and 6 restrict the explicit declaration of the mapping of local variables and the dummy arguments and dummy results. This is because the function may be invoked concurrently, with each invocation active on a subset of processors specific to that invocation, and operating on data that are mapped to that processor subset. Indeed, in an optimising implementation, the caller may well automatically arrange the mapping of the actual arguments and result according to the context, e.g. to maximise concurrency in a `FORALL`, and/or to reduce communication, taking into account the mappings of other arguments, other terms in the expression, the assignment variable, etc. Thus, a dummy argument or result may not appear in a mapping directive that fixes its location with respect to the processor array (e.g. it may not be aligned with a global variable or template, or be explicitly distributed, or given the inherit attribute, all of which would remove the caller's freedom to determine the actual's mapping as described above). The only type of mapping information that may be specified for the dummy arguments and result is their alignment with each other; this will provide useful information to the caller about their required *relative* mappings. For similar reasons, local variables may be aligned with the dummy arguments or result (either directly or through other local variables), but may not have arbitrary mappings.

Constraints 7 and 8 prevent the side effect of realignment and redistribution of data within a pure function.

The penultimate constraint prevents external I/O and file operations, whose order would be non-deterministic in the context of concurrent execution. Note that internal I/O *is* allowed, provided that it does not modify global variables or dummy arguments.

Finally, the last constraint disallows `PAUSE` and `STOP` statements. A `PAUSE` statement requires input and so is disallowed for the same reason as I/O. A `STOP` brings execution to a halt, which is a rather drastic side effect.

(*End of rationale.*)

Advice to implementors. Note that `PURE` functions may prescriptively align their dummy arguments, thus possibly causing remapping on function call. Because only alignment is involved, this cannot result in mapping data to processors that do not already store some data involved in the call.

Also note that `PURE` functions may read, but not write, distributed global data. This may be very difficult to implement on machines without shared memory. One possible implementation would be to use interrupt-driven messages to fetch data; another would be to use interprocedural analysis to detect all possible global data use in a `PURE` procedure. Some feedback from the compiler indicating such expensive access patterns would be quite valuable to serious users. (*End of advice to implementors.*)

4.3.1.2 Pure subroutine definition

The following constraints are added to Rule R1219 in Section 12.5.2.3 of the Fortran 90 standard (defining *subroutine-subprogram*):

Constraint: The *specification-part* of a pure subroutine must specify the intents of all dummy arguments except procedure arguments and arguments that have the `POINTER` attribute.

Constraint: A local variable declared in the *specification-part* or *internal-function-part* of a pure subroutine must not have the `SAVE` attribute.

Constraint: The *execution-part* or *internal-subprogram-part* of a pure subroutine must not use a dummy parameter with `INTENT(IN)`, a global variable, or an object that is storage associated with a global variable, or a subobject thereof, in the following contexts:

- As the assignment variable of an *assignment-stmt*;
- As a `DO` variable or implied `DO` variable, or as a *index-name* in a *forall-triplet-spec*;
- As an *input-item* in a *read-stmt*;
- As an *internal-file-unit* in a *write-stmt*;
- As an `IOSTAT=` or `SIZE=` specifier in an I/O statement.
- In an *assign-stmt*;
- As the *pointer-object* or *target* of a *pointer-assignment-stmt*;
- As the *expr* of an *assignment-stmt* whose assignment variable is of a derived type, or is a pointer to a derived type, that has a pointer component at any level of component selection;
- As an *allocate-object* or *stat-variable* in an *allocate-stmt* or *deallocate-stmt*, or as a *pointer-object* in a *nullify-stmt*;
- As an actual argument associated with a dummy argument with `INTENT (OUT)` or `INTENT(INOUT)` or with the `POINTER` attribute.

Constraint: Any procedure referenced in a pure subroutine, including one referenced via a defined operation or assignment, must be pure.

Constraint: A dummy argument of a pure subroutine may be explicitly aligned only with another dummy argument, and may not be explicitly distributed or given the `INHERIT` attribute.

Constraint: In a pure subroutine, a local variable may be explicitly aligned only with another local variable or a dummy argument. A local variable may not be explicitly distributed.

Constraint: In a pure subroutine, a dummy argument or local variable must not have the `DYNAMIC` attribute.

Constraint: In a pure subroutine, a global variable must not appear in a *realign-directive* or *redistribute-directive*.

Constraint: A pure subroutine must not contain a *backspace-stmt*, *close-stmt*, *endfile-stmt*, *inquire-stmt*, *open-stmt*, *print-stmt*, *rewind-stmt*, *print-stmt*, or a *read-stmt* or *write-stmt* whose *io-unit* is an *external-file-unit* or `*`.

Constraint: A pure subroutine must not contain a *pause-stmt* or *stop-stmt*.

Constraint: A pure subroutine must not contain an asterisk (`*`) in its *dummy-argument-list*.

Rationale.

The constraints for pure subroutines are based on the same principles as for pure functions, except that side effects to `INTENT(OUT)` and `INTENT(INOUT)` dummy arguments are permitted. Pointer dummy arguments are always treated as `INTENT(INOUT)`.

Pure subroutines are included to allow subroutine calls from pure procedures in a safe way, and to allow *forall-assignment*s to be defined assignments.

In addition, the last constraint disallows alternate returns in pure subroutines. These were not explicitly forbidden in pure functions, because no function can contain alternate returns. An alternate return from a pure subroutine would change the control flow in the calling routine; this was judged to be not in the spirit of pure procedures.

(*End of rationale.*)

4.3.1.3 Pure procedure interfaces

To define interface specifications for pure procedures, the following constraints are added to Rule R1204 in Section 12.3.2.1 of the Fortran 90 standard (defining *interface-body*):

Constraint: An *interface-body* of a pure procedure must specify the intents of all dummy arguments except `POINTER` and procedure arguments.

The procedure characteristics defined by an interface body must be consistent with the procedure's definition. Regarding pure procedures, this is interpreted as follows:

- A procedure that is declared pure at its definition may be declared pure in an interface body, but this is not required.
- A procedure that is not declared pure at its definition must not be declared pure in an interface body.

That is, if an interface body contains a `PURE` attribute, then the corresponding procedure definition must also contain it, though the reverse is not true. When a procedure definition with a `PURE` attribute is compiled, the compiler may check that it satisfies the necessary constraints.

4.3.2 Pure Procedure Reference

To define pure procedure references, the following extra constraint is added to Rules R1209 and R1210 in Section 12.4.1 of the Fortran 90 standard (defining *function-reference* and *call-stmt*):

Constraint: In a reference to a pure procedure, a *procedure-name actual-arg* must be the name of a pure procedure.

Rationale. This constraint ensures that the purity of a procedure cannot be undermined by allowing it to call a non-pure procedure. (*End of rationale.*)

4.3.3 Examples of Pure Procedure Usage

Pure functions may be used in expressions in FORALL statements and constructs, unlike general functions. Several examples of this are given below.

```
! This statement function is pure since it does not reference
! any other functions
REAL myexp
myexp(x) = 1 + x + x*x/2.0 + x*x*x/6.0
FORALL ( i = 1:n ) a(i) = myexp( a(i+1) )
...
! Intrinsic functions are always pure
FORALL ( i = 1:n ) a(i,i) = log( abs( a(i,i) ) )
```

Because a *forall-assignment* may be an array assignment, the pure function can have an array result. Such functions may be particularly helpful for performing row-wise or column-wise operations on an array. The next example illustrates this.

```
INTERFACE
  PURE FUNCTION f(x)
    REAL, DIMENSION(3) :: f,
    REAL, DIMENSION(3), INTENT(IN) :: x
  END FUNCTION f
END INTERFACE
REAL  v (3,10,10)
...
FORALL (i=1:10, j=1:10)  v(:,i,j) = f(v(:,i,j))
```

A limited form of MIMD parallelism can be obtained by means of branches within the pure procedure that depend on arguments associated with array elements or their subscripts when the function is called from a FORALL. This may sometimes provide an alternative to using sequences of masked FORALL or WHERE statements with their potential synchronization overhead. The next example suggests how this may be done.

```
REAL PURE FUNCTION f (x, i)
  REAL, INTENT(IN) :: x       ! associated with array element
  INTEGER, INTENT(IN) :: i    ! associated with array subscript
  IF (x > 0.0) THEN              ! content-based conditional
    f = x*x
  ELSE IF (i==1 .OR. i==n) THEN  ! subscript-based conditional
    f = 0.0
  ELSE
    f = x
  ENDIF
END FUNCTION

...

REAL a(n)
```

```
INTEGER i
  ...
FORALL (i=1:n)  a(i) = f( a(i), i)
```

Because pure procedures have no constraints on their internal control flow (except that they may not use the STOP statement), they also provide a means for encapsulating more complex operations than could otherwise be nested within a FORALL. For example, the fragment below performs an iterative algorithm on every element of an array. Note that different amounts of computation may be required for different inputs. Some machines may not be able to take advantage of this flexibility.

```
PURE INTEGER FUNCTION iter(x)
  COMPLEX, INTENT(IN) :: x
  COMPLEX xtmp
  INTEGER i
  i = 0
  xtmp = -x
  DO WHILE (ABS(xtmp).LT.2.0 .AND. i.LT.1000)
    xtmp = xtmp * xtmp - x
    i = i + 1
  END DO
  iter = i
END FUNCTION

  ...
FORALL (i=1:n, j=1:m)  ix(i,j) = iter(CMPLX(a+i*da,b+j*db))
```

4.3.4 Comments on Pure Procedures

Rationale.

The constraints for a pure procedure guarantee freedom from side-effects, thus ensuring that it can be invoked concurrently at each "element" of an array (where an "element" may itself be a data structure, including an array).

The constraints on pure procedures may appear complicated, but it is not necessary for a programmer to be intimately familiar with them. ¿From the programmer's point of view, these constraints can be summarized as follows: a pure procedure must not contain any operation that could conceivably result in an assignment or pointer assignment to a global variable or INTENT (IN) dummy argument, or perform any I/O or STOP operation. Note the use of the word *conceivably*; it is not sufficient for a pure procedure merely to be side-effect free *in practice*. For example, a function that contains an assignment to a global variable but in a branch that is not executed in any invocation of the function is nevertheless not a pure function. The exclusion of functions of this nature is unavoidable if strict compile-time checking is to be used. In the choice between compile-time checking and flexibility, the HPF committee decided in favor of enhanced checking.

It is expected that most library procedures will conform to the constraints required of pure procedures (by the very nature of library procedures), and so can be declared pure and referenced in FORALL statements and constructs and within user-defined pure procedures. It is also anticipated that most library procedures will not reference global data, whose use may sometimes inhibit concurrent execution.

The constraints on pure procedures are limited to those necessary to check statically for freedom from side effects, processor independence, and for lack of saved internal state. Subject to these restrictions, maximum functionality has been preserved in the definition of pure procedures. This has been done to make function calls in `FORALL` as widely available as possible, and so that quite general library procedures can be classified as pure.

A drawback of this flexibility is that pure procedures permit certain features whose use may hinder, and in the worst case prevent, concurrent execution in `FORALL` (that is, such references may have to be implemented by sequentialization). Foremost among these features are the access of global data, particularly distributed global data, and the fact that the arguments and, for a pure function, the result may be pointers or data structures with pointer components, including recursive data structures such as lists and trees. The programmer should be aware of the potential performance penalties of using such features.

(*End of rationale.*)

4.4 The INDEPENDENT Directive

The `INDEPENDENT` directive can precede an indexed `DO` loop or `FORALL` statement or construct. It asserts to the compiler that the operations in the following `FORALL` statement or construct or iterations in the following `DO` loop may be executed independently—that is, in any order, or interleaved, or concurrently—without changing the semantics of the program.

The `INDEPENDENT` directive precedes the `DO` loop or `FORALL` for which it is asserting behavior, and is said to apply to that loop or `FORALL`. The syntax of the `INDEPENDENT` directive is

H413	*independent-directive*	**is**	`INDEPENDENT` [, *new-clause*]
H414	*new-clause*	**is**	`NEW` (*variable-list*)

Constraint: The first non-comment line following an *independent-directive* must be a *do-stmt*, *forall-stmt*, or a *forall-construct*.

Constraint: If the first non-comment line following an *independent-directive* is a *do-stmt*, then that statement must contain a *loop-control* option containing a *do-variable*.

Constraint: If the `NEW` option is present, then the directive must apply to a `DO` loop.

Constraint: A *variable* named in the `NEW` option or any component or element thereof must not:

- Be a pointer or dummy argument; nor
- Have the `SAVE` or `TARGET` attribute.

Rationale. The second constraint means that an `INDEPENDENT` directive loop cannot be applied to a `WHILE` or a simple `DO` (i.e. a "do forever"). An `INDEPENDENT` in such cases could only indicate loops with zero or one trips, and the confusion factor in those cases was felt to outweigh the possible benefits. (*End of rationale.*)

When applied to a `DO` loop, an `INDEPENDENT` directive is an assertion by the programmer that no iteration can affect any other iteration, either directly or indirectly. The following operations define such interference:

- Any two operations that assign to the same atomic object (defined in Section 4.1.2) interfere with each other. (Note the `NEW` clause below, however.)
- An operation that assigns to an atomic object interferes with any operation that uses the value of that object. (Note the `NEW` clause below, however.)

  Rationale. These are the classic Bernstein [5] conditions to enable parallel execution. Note that two assignments *of the same value* to a variable interfere with each other and thus an `INDEPENDENT` loop with such assignments is not HPF-conforming. This is not allowed because such overlapping assignments are difficult to support on some hardware, and because the given definition was felt to be conceptually clearer. Similarly, it is not HPF-conforming to assert that assignment of multiple values to the same location is `INDEPENDENT`, even if the program logically can accept any of the possible values. In this case, both the "conceptually clearer" argument and the desire to avoid nondeterministic behavior favored the given solution. (*End of rationale.*)

- An `ALLOCATE` statement, `DEALLOCATE` statement, `NULLIFY` statement or pointer assignment statement interferes with any other access, pointer assignment, allocation, deallocation, or nullification of the same pointer. In addition, a `DEALLOCATE` statement interferes with any other use of or assignment to the object which is deallocated.

  Rationale. These constraints extend Bernstein's conditions to pointers. Because a Fortran 90 pointer is an alias to a section of memory rather than a first-class data type, a bit more precision is needed than for other variables. (*End of rationale.*)

- Any transfer of control to a branch target statement outside the body of the loop interferes with all other operations in the loop.
- Any execution of an `EXIT`, `STOP`, or `PAUSE` statement interferes with all other operations in the loop.

  Rationale. Branching (by `GOTO` or `ERR=` branches in I/O statements) implies that some iterations of the loop are not executed, which is drastic interference with those computations. The same is true for `EXIT` and the other statements. Note that these conditions do not restrict procedure calls in `INDEPENDENT` loops, except to disallow taking alternate returns to statements outside the loop. (*End of rationale.*)

- A `READ` operation assigns to the objects in its *input-item-list*; a `WRITE` or `PRINT` operation uses the values of the objects on its *output-item-list*. I/O operations may interfere with other operations (including other I/O operations) as per the conditions above.

- An internal READ operation uses its internal file; an internal WRITE operation assigns to its internal file. These uses and assignments may interfere with other operations as outlined above.

- Any two file I/O operations except INQUIRE associated with the same file or unit interfere with each other. Two INQUIRE operations do not interfere with each other; however, an INQUIRE operation interferes with any other I/O operation associated with the same file.

  Rationale. Because Fortran carefully defines the file position after a data transfer or file positioning statement, these operations affect the global state of a program. (Note that file position is defined even for direct access files.) Multiple non-advancing data transfer statements affect the file position in ways similar to multiple assignments of the same value to a variable, and is disallowed for the same reason. Multiple OPEN and CLOSE operations affect the status of files and units, which is another global side effect. INQUIRE does not affect the file status, and therefore does not affect other inquiries. However, other file operations may affect the properties reported by INQUIRE. (*End of rationale.*)

- Any data realignment or redistribution performed in the loop interferes with any access to or any other realignment of the same data.

  Rationale. REALIGN and REDISTRIBUTE may change the processor storing a particular array element, which interferes with any assignment or use of that element. Similarly, multiple remapping operations may cause the same element to be stored in multiple locations. (*End of rationale.*)

If a procedure is called from within an INDEPENDENT loop or FORALL, then any local variables in that procedure are considered distinct on each call unless they have the SAVE attribute. This is consistent with the Fortran 90 standard. Therefore, uses of local variables on calls from different iterations do not cause interference as defined above.

Advice to implementors. A legal Fortran 90 implementation can often avoid creating distinct storage for locals on every call. The same is true for an HPF implementation; however, such an implementation must still interpret INDEPENDENT in the same way. If locals are not allocated unique storage locations on every call, then the INDEPENDENT loop must be serialized to respect these semantics (or other techniques must be used for the purpose). (*End of advice to implementors.*)

Note that all of these describe interfering behavior; they do not disallow specific syntax. Statements that appear to violate one or more of these restrictions are allowed in an INDEPENDENT loop, if they are not executed due to control flow. These restrictions allow an INDEPENDENT loop to be executed safely in parallel if computational resources are available. The directive is purely advisory and a compiler is free to ignore it if it cannot make use of the information.

Advice to implementors. Although the restrictions allow safe parallel implementation of INDEPENDENT loops, they do not imply that this will be profitable (or even possible) on all architectures or all programs. For example,

- An INDEPENDENT loop may call a routine with explicitly mapped local variables. The implementation must then either implement the mapping (which may require serializing the calls) or override the explicit directives (which may surprise the user).
- An INDEPENDENT loop may have very different behavior on every iteration. For example,

  ```
  !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
  ```

  This poses obvious problems for implementations on SIMD machines.

In all cases, it is the implementation's responsibility to produce correct behavior, which may in turn limit optimization. It is recommended that implementations provide some feedback if an INDEPENDENT assertion may be ignored. (*End of advice to implementors.*)

The NEW option modifies the INDEPENDENT directive and all surrounding INDEPENDENT directives by asserting that those assertions would be true *if* new objects were created for the named variables for each iteration of the DO loop. Thus, variables named in the *new-clause* behave as if they were private to the body of the DO loop. More formally, it asserts that the remainder of program execution is unaffected if all variables in the *variable-list* and any variables associated with them were to become undefined immediately before execution of every iteration of the loop, and also become undefined immediately after the completion of each iteration of the loop.

Advice to implementors.

The wording here is similar to the treatment of realignment through pointers in Section 3.6. As with that section, it may be reworded if HPF directives are absorbed as actual Fortran statements.

(*End of advice to implementors.*)

Rationale. NEW variables provide the means to declare temporaries in INDEPENDENT loops. Without this feature, many conceptually independent loops would need substantial rewriting (including expansion of scalars into arrays) to meet the rather strict requirements described above. Note that a temporary need only be declared NEW at the innermost lexical level at which it is assigned, since all enclosing INDEPENDENT assertions must take that NEW into account. Note also that index variables for nested DO loops must be declared NEW; the alternative was to limit the scope of an index variable to the loop itself, which changes Fortran semantics. FORALL indices, however, are restricted by the semantics of the FORALL; they require no NEW declarations. (*End of rationale.*)

Advice to users. Section 4.4.1 contains several examples of the syntax and semantics of INDEPENDENT applied to DO loops. (*End of advice to users.*)

The interpretation of INDEPENDENT for FORALL is similar to that for DO: it asserts that no combination of the indexes that INDEPENDENT applies assigns to an atomic storage unit that is read by another combination. (Note that an HPF FORALL statement or construct does not allow exits from the construct, etc.) A DO and a FORALL with the same body are equivalent if they both have the INDEPENDENT directive. This is illustrated in Section 4.4.2.

4.4.1 Examples of INDEPENDENT

```
!HPF$ INDEPENDENT
DO i = 2, 99
  a(i) = b(i-1) + b(i) + b(i+1)
END DO
```

This is one of the simplest examples of an INDEPENDENT loop. (For simplicity, all examples in this section assume there is no storage or sequence association between any variables used in the code.) Every iteration assigns to a different location in the a array, thus satisfying the first condition above. Since no elements of a are used on the right-hand side, no location that is assigned in the loop is also read, thus satisfying the second condition. Note, however, that many elements of b are used repeatedly; this is allowed by the definition of INDEPENDENT. The other conditions relate to constructs not used in the loop. In this example, the assertion is true regardless of the values of the variables involved.

```
!HPF$ INDEPENDENT
FORALL ( i=2:n ) a(i) = b(i-1) + b(i) + b(i+1)
```

This example is equivalent in all respects to the first example.

```
!HPF$ INDEPENDENT
DO i=1, 100
  a(p(i)) = b(i)
END DO
```

This INDEPENDENT directive asserts that the array p does not have any repeated entries (else they would cause interference when a was assigned). The DO loop is therefore equivalent to the Fortran 90 statement

```
a(p(1:100)) = b(1:100)
```

```
!HPF$ INDEPENDENT, NEW (i2)
DO i1 = 1,n1
 !HPF$ INDEPENDENT, NEW (i3)
 DO i2 = 1,n2
  !HPF$ INDEPENDENT, NEW (i4)
  DO i3 = 1,n3
   DO i4 = 1,n4   ! The inner loop is NOT independent!
    a(i1,i2,i3) = a(i1,i2,i3) + b(i1,i2,i4)*c(i2,i3,i4)
   END DO
  END DO
 END DO
END DO
```

The inner loop is not independent because each element of a is assigned repeatedly. However, the three outer loops are independent because they access different elements of a. The NEW clauses are required, since the inner loop indices are assigned and used in different iterations of the outermost loops.

```
!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
```

Without the NEW option on the j loop, neither loop would be independent, because an interleaved execution of loop iterations might cause other values of vl, vr, ul, and ur to be used in the assignment of $p(i, j)$ than those computed in the same iteration of the loop. The NEW option, however, specifies that this is not true if distinct storage units are used in each iteration of the loop. Using this implementation makes iterations of the loops independent of each other. Note that there is no interference due to accesses of the array p because of the stride of the DO loop (i.e. i and j are always even, therefore $i-1$, etc. are always odd.)

```
!HPF$ INDEPENDENT
      DO i = 1, 10
        WRITE (iounit(i),100) a(i)
      END DO
100   FORMAT ( F10.4 )
```

If $iounit(i)$ evaluates to a different value for every $i \in \{1, \ldots, 10\}$, then the loop writes to a different I/O unit (and thus a different file) on every iteration. The loop is then properly described as independent. On the other hand, if $iounit(i) = 5$ for all i, then the assertion is in error and the loop is not HPF-conforming.

4.4.2 Visualization of INDEPENDENT Directives

Graphically, the INDEPENDENT directive can be visualized as eliminating edges from a precedence graph representing the program. Figure 4.1 shows some of the dependences that may normally be present in a DO and a FORALL. (Most of the transitive dependences are not shown.) An arrow from a left-hand side node (for example, "lhsa(1)") to a right-hand side node ("rhsb(1)") means that the right-hand side computation might use values assigned in the left-hand side node; thus the right-hand side must be computed after the left-hand side completes its store. Similarly, an arrow from a right-hand side node to a left-hand side node means that the left-hand side may overwrite a value needed by the right-hand side computation, again forcing an ordering. Edges from the "BEGIN" and to the "END" nodes represent control dependences. The INDEPENDENT directive asserts that the only dependences that a compiler need enforce are those in Figure 4.2. That is, the programmer

```
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
```

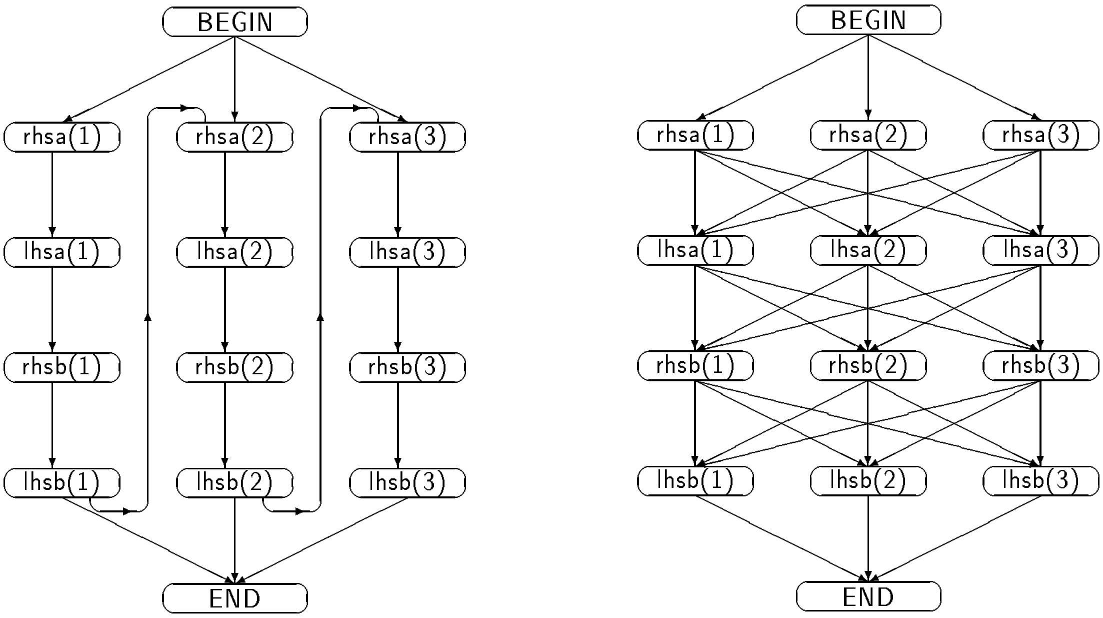


Figure 4.1: Dependences in `DO` and `FORALL` without `INDEPENDENT` assertions

```
!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
```

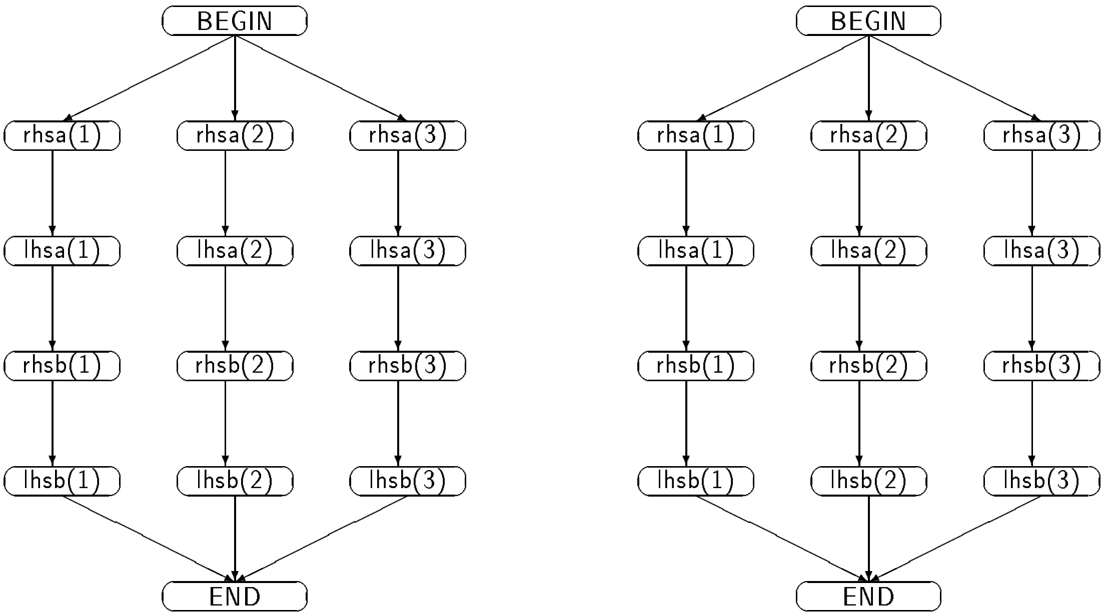


Figure 4.2: Dependences in `DO` and `FORALL` with `INDEPENDENT` assertions

who uses `INDEPENDENT` is certifying that if the compiler enforces only these edges, then the resulting program will be equivalent to the one in which all the edges are present. Note that the set of asserted dependences is identical for `INDEPENDENT DO` and `FORALL` constructs.

The compiler is justified in producing a warning if it can prove that one of these assertions is incorrect. It is not required to do so, however. A program containing any false assertion of this type is not HPF-conforming, thus is not defined by HPF, and the compiler may take any action it deems appropriate.

Section 5

Intrinsic and Library Procedures

HPF includes Fortran 90's intrinsic procedures. It also adds new intrinsic procedures in two categories: system inquiry intrinsic functions and computational intrinsic functions.

The definitions of two Fortran 90 intrinsic functions, `MAXLOC` and `MINLOC`, are extended by the addition of an optional `DIM` argument.

In addition to the new intrinsic functions, HPF defines a library module, `HPF_LIBRARY`, that must be provided by vendors of any full HPF implementation.

This description of HPF intrinsic and library procedures follows the form and conventions of Section 13 of the Fortran 90 standard. The material of Sections 13.1, 13.2, 13.3, 13.5.7, 13.8.1, 13.8.2, 13.9, and 13.10 is applicable to the HPF intrinsic and library procedures and to their descriptions in this section of the HPF document.

5.1 Notation

In the examples of this section, `T` and `F` are used to denote the logical values true and false.

5.2 System Inquiry Intrinsic Functions

In a multi-processor implementation, the processors may be arranged in an implementation-dependent multi-dimensional processor array. The system inquiry functions return values related to this underlying machine and processor configuration, including the size and shape of the underlying processor array. `NUMBER_OF_PROCESSORS` returns the total number of processors available to the program or the number of processors available to the program along a specified dimension of the processor array. `PROCESSORS_SHAPE` returns the shape of the processor array.

The values returned by the system inquiry intrinsic functions remain constant for the duration of one program execution. Thus, `NUMBER_OF_PROCESSORS` and `PROCESSORS_SHAPE` have values that are restricted expressions and may be used wherever any other Fortran 90 restricted expression may be used. In particular, `NUMBER_OF_PROCESSORS` may be used in a specification expression.

The values of system inquiry functions may not occur in initialization expressions, because they may not be assumed to be constants. In particular, HPF programs may be compiled to run on machines whose configurations are not known at compile time.

Note that the system inquiry functions query the physical machine, and have nothing to do with any `PROCESSORS` directive that may occur.

Advice to users. SIZE(PROCESSORS_SHAPE()) returns the rank of the processor array. References to system inquiry functions may occur in array declarations and in HPF directives, as in:

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

(*End of advice to users.*)

5.3 Computational Intrinsic Functions

HPF adds one new intrinsic function, ILEN, which computes the number of bits needed to store an integer value. HPF also generalizes the Fortran 90 MAXLOC and MINLOC intrinsic functions with an optional DIM parameter, for finding the locations of maximum or minimum elements along a given dimension.

The HPF and the Fortran 90 intrinsic functions MAXLOC and MINLOC have a required first argument, ARRAY. In HPF, these functions have an optional second argument, DIM of type integer, and an optional third argument, MASK of type logical. The Fortran 90 intrinsic functions MAXLOC and MINLOC have only one optional argument, MASK of type logical.

Thus, an invocation with two arguments in Fortran 90, the second being the mask argument, might be interpreted incorrectly by an HPF compiler. The type of DIM must be integer and the type of MASK must be logical, however, and an HPF implementation is required to correctly distinguish by the type of the second actual argument, in invocations with two arguments present, between these possibilities.

5.4 Library Procedures

The mapping inquiry subroutines and computational functions described in this section are available in the HPF library module, HPF_LIBRARY. Use of these procedures must be accompanied by an appropriate USE statement in each scoping unit in which they are used. They are not intrinsic.

5.4.1 Mapping Inquiry Subroutines

HPF provides data mapping directives that are advisory in nature. The mapping inquiry subroutines allow the program to determine the actual mapping of an array at run time. It may be especially important to know the exact mapping when an EXTRINSIC subprogram is invoked. For these reasons, HPF includes mapping inquiry subroutines which describe how an array is actually mapped onto a machine. To keep the number of routines small, the inquiry procedures are structured as subroutines with optional INTENT (OUT) arguments.

5.4.2 Bit Manipulation Functions

The HPF library includes three elemental bit-manipulation functions. LEADZ computes the number of leading zero bits in an integer's representation. POPCNT counts the number of one bits in an integer. POPPAR computes the parity of an integer.

5.4.3 Array Reduction Functions

HPF adds additional array reduction functions that operate in the same manner as the Fortran 90 SUM and ANY intrinsic functions. The new reduction functions are IALL, IANY, IPARITY, and PARITY, which correspond to the commutative, associative binary operations IAND, IOR, IEOR, and .NEQV. respectively.

In the specifications of these functions, the terms "XXX reduction" are used, where XXX is one of the binary operators above. These are defined by means of an example. The IAND reduction of all the elements of array for which the corresponding element of mask is true is the scalar integer computed in result by

```
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
```

Here, n is the rank of array and IAND_IDENTITY_ELEMENT is the integer which has all bits equal to one. (The interpretation of an integer as a sequence of bits is given in Section 13.5.7 of the Fortran 90 standard.) The other three reductions are similarly defined. The identity elements for IOR and IEOR are zero. The identity element for PARITY is .FALSE.

5.4.4 Array Combining Scatter Functions

These are all generalized array reduction functions in which completely general, but nonoverlapping, subsets of array elements can be combined. There is a corresponding scatter function for each of the twelve reduction operation in the language. The way the elements of the source array are associated with the elements of the result is described in this section; the method of combining their values is described in the specifications of the individual functions in Section 5.7.

These functions all have the form

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

The allowed values of XXX are ALL, ANY, COPY, COUNT, IALL, IANY, IPARITY, MAXVAL, MINVAL, PARITY, PRODUCT, and SUM. The number of INDX arguments must equal the rank of BASE. Except for COUNT_SCATTER, ARRAY and BASE are arrays of the same type. For COUNT_SCATTER, ARRAY is of type logical and BASE is of type integer. The argument MASK is logical, and the INDX arrays are integer. ARRAY, MASK, and all the INDX arrays are conformable. MASK is optional. (For ALL_SCATTER, ANY_SCATTER, COUNT_SCATTER,and PARITY_SCATTER, the ARRAY must be logical. These functions do not have an optional MASK argument. To conform with the conventions of the F90 standard, the required ARRAY argument to these functions is called MASK in their specifications in Section 5.7.) The result has the same type, kind type parameter, and shape as BASE.

For every element a in `ARRAY` there is a corresponding element in each of the `INDX` arrays. Let s_1 be the value of the element of `INDX1` that is indexed by the same subscripts as element a of `ARRAY`. More generally, for each $j = 1, 2, ..., n$, let s_j be the value of the element of `INDXj` that corresponds to element a in `ARRAY`, where n is the rank of `BASE`. The integers $s_j, j = 1, ..., n$, form a subscript selecting an element of `BASE`: `BASE(`$s_1, s_2, ..., s_n$`)`.

Thus the `INDX` arrays establish a mapping from all the elements of `ARRAY` onto selected elements of `BASE`. Viewed in the other direction, this mapping associates with each element b of `BASE` a set S of elements from `ARRAY`.

Because `BASE` and the result are conformable, for each element of `BASE` there is a corresponding element of the result.

If S is empty, then the element of the result corresponding to the element b of `BASE` has the same value as b.

If S is non-empty, then the elements of S will be combined with element b to produce an element of the result. The particular means of combining these values is described in the result value section of the specification of the routine below. As an example, for `SUM_SCATTER`, if the elements of S are $a_1, ..., a_m$, then the element of the result corresponding to the element b of `BASE` is the result of evaluating `SUM((/`$a_1, a_2, \ldots, a_m, b$`/))`.

Note that, since a scalar is conformable with any array, a scalar may be used in place of an `INDX` array, in which case one hyperplane of the result is selected. See the example below.

If the optional, final `MASK` argument is present, then only the elements of `ARRAY` in positions for which `MASK` is true participate in the operation. All other elements of `ARRAY` and of the `INDX` arrays are ignored and cannot have any influence on any element of the result.

For example, if

`A` is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$; `B` is the array $\begin{bmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & -8 & -9 \end{bmatrix}$;

`I1` is the array $\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 1 \\ 3 & 2 & 1 \end{bmatrix}$; `I2` is the array $\begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 2 \\ 1 & 1 & 1 \end{bmatrix}$

then

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

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

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

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

If `A` is the array $\begin{bmatrix} 10 & 20 & 30 & 40 & -10 \end{bmatrix}$, `B` is the array $\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$,
and `IND` is the array $\begin{bmatrix} 3 & 2 & 2 & 1 & 1 \end{bmatrix}$,
then `SUM_SCATTER(A, B, IND, MASK=(A .GT. 0))` is $\begin{bmatrix} 41 & 52 & 13 & 4 \end{bmatrix}$.

5.4.5 Array Prefix and Suffix Functions

In a scan of a vector, each element of the result is a function of the elements of the vector that precede it (for a prefix scan) or that follow it (for a suffix scan). These functions provide scan operations on arrays and subarrays. The functions all have the form

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

The allowed values of `XXX` are `ALL`, `ANY`, `COPY`, `COUNT`, `IALL`, `IANY`, `IPARITY`, `MAXVAL`, `MINVAL`, `PARITY`, `PRODUCT`, and `SUM`.

When comments below apply to both prefix and suffix forms of the routines, we will refer to them as `YYYFIX` functions.

The arguments `DIM`, `MASK`, `SEGMENT`, and `EXCLUSIVE` are optional. The `COPY_YYYFIX` functions do not have `MASK` or `EXCLUSIVE` arguments. The `ALL_YYYFIX`, `ANY_YYYFIX`, `COUNT_YYYFIX`, and `PARITY_YYYFIX` functions do not have `MASK` arguments. Their `ARRAY` argument must be of type logical; it is denoted `MASK` in their specifications in Section 5.7.

The arguments `MASK` and `SEGMENT` must be of type logical. `SEGMENT` must have the same shape as `ARRAY`. `MASK` must be conformable with `ARRAY`. `EXCLUSIVE` is a logical scalar. `DIM` is a scalar integer between one and the rank of `ARRAY`.

> **Result Value.** The result has the same shape as `ARRAY`, and, with the exception of `COUNT_YYYFIX`, the same type and kind type parameter as `ARRAY`. (The result of `COUNT_YYYFIX` is default integer.)
>
> In every case, every element of the result is determined by the values of certain selected elements of `ARRAY` in a way that is specific to the particular function and is described in its specification. The optional arguments affect the selection of elements of `ARRAY` for each element of the result; the selected elements of `ARRAY` are said to contribute to the result element. This section describes fully which elements of `ARRAY` contribute to a given element of the result.
>
> If no elements of `ARRAY` are selected for a given element of the result, that result element is set to a default value that is specific to the particular function and is described in its specification.
>
> For any given element r of the result, let a be the corresponding element of `ARRAY`. Every element of `ARRAY` contributes to r unless disqualified by one of the following rules.
>
> 1. If the function is `XXX_PREFIX`, no element that follows a in the array element ordering of `ARRAY` contributes to r. If the function is `XXX_SUFFIX`, no element that precedes a in the array element ordering of `ARRAY` contributes to r.
> 2. If the `DIM` argument is provided, an element z of `ARRAY` does not contribute to r unless all its indices, excepting only the index for dimension `DIM`, are the same as the corresponding indices of a. (It follows that if the `DIM` argument is

omitted, then ARRAY, MASK, and SEGMENT are processed in array element order, as if temporarily regarded as rank-one arrays. If the DIM argument is present, then a family of completely independent scan operations are carried out along the selected dimension of ARRAY.)

3. If the MASK argument is provided, an element z of ARRAY contributes to r only if the element of MASK corresponding to z is true. (It follows that array elements corresponding to positions where the MASK is false do not contribute anywhere to the result. However, the result is nevertheless defined at all positions, even positions where the MASK is false.)

4. If the SEGMENT argument is provided, an element z of ARRAY does not contribute if there is some intermediate element w of ARRAY, possibly z itself, with all of the following properties:

   (a) If the function is XXX_PREFIX, w does not precede z but does precede a in the array element ordering; if the function is XXX_SUFFIX, w does not follow z but does follow a in the array element ordering;

   (b) If the DIM argument is present, all the indices of w, excepting only the index for dimension DIM, are the same as the corresponding indices of a; and

   (c) The element of SEGMENT corresponding to w does not have the same value as the element of SEGMENT corresponding to a. (In other words, z can contribute only if there is an unbroken string of SEGMENT values, all alike, extending from z through a.)

5. If the EXCLUSIVE argument is provided and is true, then a itself does not contribute to r.

These general rules lead to the following important cases:

Case (i): If ARRAY has rank one, element i of the result of XXX_PREFIX(ARRAY) is determined by the first i elements of ARRAY; element SIZE(ARRAY) $- i + 1$ of the result of XXX_SUFFIX(ARRAY) is determined by the last i elements of ARRAY.

Case (ii): If ARRAY has rank greater than one, then each element of the result of XXX_PREFIX(ARRAY) has a value determined by the corresponding element a of the ARRAY and all elements of ARRAY that precede a in array element order. For XXX_SUFFIX, a is determined by the elements of ARRAY that correspond to or follow a in array element order.

Case (iii): Each element of the result of XXX_PREFIX(ARRAY,MASK=MASK) is determined by selected elements of ARRAY, namely the corresponding element a of the ARRAY and all elements of ARRAY that precede a in array element order, but an element of ARRAY may contribute to the result only if the corresponding element of MASK is true. If this restriction results in selecting no array elements to contribute to some element of the result, then that element of the result is set to the default value for the given function.

Case (iv): Each element of the result of XXX_PREFIX(ARRAY,DIM=DIM) is determined by selected elements of ARRAY, namely the corresponding element a of the ARRAY and all elements of ARRAY that precede a along dimension

DIM; for example, in SUM_PREFIX(A(1:N,1:N), DIM=2), result element (i_1, i_2) could be computed as SUM(A(i_1,1 : i_2)). More generally, in SUM_PREFIX(ARRAY, DIM), result element $i_1, i_2, \ldots, i_{DIM}, \ldots, i_n$ could be computed as SUM(ARRAY($i_1, i_2, \ldots, :i_{DIM}, ..., i_n$)) . (Note the colon before i_{DIM} in that last expression.)

Case (v): If ARRAY has rank one, then element i of the result of XXX_PREFIX(ARRAY, EXCLUSIVE=.TRUE.) is determined by the first $i - 1$ elements of ARRAY.

Case (vi): The options may be used in any combination.

Advice to users. A new segment begins at every *transition* from false to true or true to false; thus a segment is indicated by a maximal contiguous subsequence of like logical values:

```
(/T,T,T,F,T,F,F,F,T,F,F,T/)
  ----- - - ----- - --- -    seven segments
```

(*End of advice to users.*)

Rationale.

One existing library delimits the segments by indicating the *start* of each segment. Another delimits the segments by indicating the *stop* of each segment. Each method has its advantages. There is also the question of whether this convention should change when performing a suffix rather than a prefix. HPF adopts the symmetric representation above. The main advantages of this representation are:

(A) It is symmetrical, in that the same segment specifier may be meaningfully used for prefix and suffix without changing its interpretation (start versus stop).

(B) The start-bit or stop-bit representation is easily converted to this form by using PARITY_PREFIX or PARITY_SUFFIX. These might be standard idioms for a compiler to recognize:

```
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))
```

(*End of rationale.*)

Examples. The examples below illustrate all possible combinations of optional arguments for SUM_PREFIX. The default value for SUM_YYYFIX is zero.

Case (i): SUM_PREFIX((/1,3,5,7/)) is $\begin{bmatrix} 1 & 4 & 9 & 16 \end{bmatrix}$.

Case (ii): If B is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$,

SUM_PREFIX(B) is the array $\begin{bmatrix} 1 & 14 & 30 \\ 5 & 19 & 36 \\ 12 & 27 & 45 \end{bmatrix}$.

Case (iii): If `A` is the array $\begin{bmatrix} 3 & 5 & -2 & -1 & 7 & 4 & 8 \end{bmatrix}$,
then `SUM_PREFIX(A, MASK = A .LT. 6)` is $\begin{bmatrix} 3 & 8 & 6 & 5 & 5 & 9 & 9 \end{bmatrix}$.

Case (iv): If `B` is the array $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, then `SUM_PREFIX(B, DIM=1)` is the array $\begin{bmatrix} 1 & 2 & 3 \\ 5 & 7 & 9 \\ 12 & 15 & 18 \end{bmatrix}$ and `SUM_PREFIX(B, DIM=2)` is the array $\begin{bmatrix} 1 & 3 & 6 \\ 4 & 9 & 15 \\ 7 & 15 & 24 \end{bmatrix}$.

Case (v): `SUM_PREFIX((/1,3,5,7/), EXCLUSIVE=.TRUE.)` is $\begin{bmatrix} 0 & 1 & 4 & 9 \end{bmatrix}$.

Case (vi): If `B` is the array $\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix}$, `M` is the array $\begin{bmatrix} T & T & T & T & T \\ F & F & T & T & T \\ T & F & T & F & F \end{bmatrix}$, and `S` is the array $\begin{bmatrix} T & T & F & F & F \\ F & T & T & F & F \\ T & T & T & T & T \end{bmatrix}$, then:

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

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

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

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

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

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

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

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

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

SUM_PREFIX(B, MASK=M, SEGMENT=S, EXCLUSIVE=.FALSE.) is

$$\begin{bmatrix} 1 & 13 & 3 & 4 & 5 \\ 0 & 13 & 8 & 13 & 15 \\ 11 & 13 & 21 & 0 & 0 \end{bmatrix}.$$

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

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

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

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

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

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

5.4.6 Array Sorting Functions

HPF includes procedures for sorting multidimensional arrays. These are structured as functions that return sorting permutations. An array can be sorted along a given axis, or the whole array may be viewed as a sequence in array element order. The sorts are stable, allowing for convenient sorting of structures by major and minor keys.

5.5 Generic Intrinsic and Library Procedures

For all of the intrinsic and library procedures, the arguments shown are the names that must be used for keywords when using the keyword form for actual arguments. Many of the argument keywords have names that are indicative of their usage, as is the case in Fortran 90. See Section 13.10 of the standard.

5.5.1 System inquiry intrinsic functions

`NUMBER_OF_PROCESSORS(DIM)`	The number of executing processors
Optional `DIM`	
`PROCESSORS_SHAPE()`	The shape of the executing processor array

5.5.2 Array location intrinsic functions

`MAXLOC(ARRAY, DIM, MASK)`	Location of a maximum value in an array
Optional `DIM, MASK`	
`MINLOC(ARRAY, DIM, MASK)`	Location of a minimum value in an array
Optional `DIM, MASK`	

5.5.3 Mapping inquiry subroutines

`HPF_ALIGNMENT(ALIGNEE, LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, &`
`DYNAMIC, NCOPIES)`
Optional `LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES`

`HPF_TEMPLATE(ALIGNEE, TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO, &`
`NUMBER_ALIGNED, DYNAMIC)`
Optional `TEMPLATE_RANK, LB, UB, AXIS_TYPE, AXIS_INFO,`
`NUMBER_ALIGNED, DYNAMIC`

`HPF_DISTRIBUTION(DISTRIBUTEE, AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, &`
`PROCESSORS_SHAPE)`
Optional `AXIS_TYPE, AXIS_INFO, PROCESSORS_RANK, PROCESSORS_SHAPE`

5.5.4 Bit manipulation functions

`ILEN(I)`	Bit length (intrinsic)
`LEADZ(I)`	Leading zeros
`POPCNT(I)`	Number of one bits
`POPPAR(I)`	Parity

5.5.5 Array reduction functions

`IALL(ARRAY, DIM, MASK)`	Bitwise logical AND reduction
Optional `DIM, MASK`	
`IANY(ARRAY, DIM, MASK)`	Bitwise logical OR reduction
Optional `DIM, MASK`	
`IPARITY(ARRAY, DIM, MASK)`	Bitwise logical EOR reduction
Optional `DIM, MASK`	
`PARITY(MASK, DIM)`	Logical EOR reduction
Optional `DIM`	

5.5.6 Array combining scatter functions

```
ALL_SCATTER(MASK, BASE, INDX1 ..., INDXn)
ANY_SCATTER(MASK, BASE, INDX1, ..., INDXn)
COPY_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
     Optional MASK
COUNT_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
     Optional MASK
IALL_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
     Optional MASK
IANY_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
     Optional MASK
IPARITY_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
     Optional MASK
IALL_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
     Optional MASK
MAXVAL_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
     Optional MASK
MINVAL_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
     Optional MASK
PARITY_SCATTER(MASK, BASE, INDX1, ..., INDXn)
PRODUCT_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
     Optional MASK
SUM_SCATTER(ARRAY, BASE, INDX1, ..., INDXn, MASK)
     Optional MASK
```

5.5.7 Array prefix and suffix functions

```
ALL_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
     Optional DIM, SEGMENT, EXCLUSIVE
ALL_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
     Optional DIM, SEGMENT, EXCLUSIVE
ANY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
     Optional DIM, SEGMENT, EXCLUSIVE
ANY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
     Optional DIM, SEGMENT, EXCLUSIVE
```

COPY_PREFIX(ARRAY, DIM, SEGMENT)
Optional DIM, SEGMENT
COPY_SUFFIX(ARRAY, DIM, SEGMENT)
Optional DIM, SEGMENT
COUNT_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
Optional DIM, SEGMENT, EXCLUSIVE
COUNT_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
Optional DIM, SEGMENT, EXCLUSIVE
IALL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
IALL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
IANY_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
IANY_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
IPARITY_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
IPARITY_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
MAXVAL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
MAXVAL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
MINVAL_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
MINVAL_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
PARITY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
Optional DIM, SEGMENT, EXCLUSIVE
PARITY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)
Optional DIM, SEGMENT, EXCLUSIVE
PRODUCT_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
PRODUCT_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
SUM_PREFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE
SUM_SUFFIX(ARRAY, DIM, MASK, SEGMENT, EXCLUSIVE)
Optional DIM, MASK, SEGMENT, EXCLUSIVE

5.5.8 Array sort functions

GRADE_DOWN(ARRAY,DIM) Optional DIM	Permutation that sorts into descending order
GRADE_UP(ARRAY,DIM) Optional DIM	Permutation that sorts into ascending order

5.6 Specifications of Intrinsic Procedures

5.6.1 ILEN(I)

Description. Returns one less than the length, in bits, of the two's-complement representation of an integer.

Class. Elemental function.

Argument. `I` must be of type integer.

Result Type and Type Parameter. Same as `I`.

Result Value. If `I` is nonnegative, `ILEN(I)` has the value $\lceil \log_2(\mathtt{I}+1) \rceil$; if `I` is negative, `ILEN(I)` has the value $\lceil \log_2(-\mathtt{I}) \rceil$.

Examples. `ILEN(4) = 3`. `ILEN(-4) = 2`. `2**ILEN(N-1)` rounds `N` up to a power of 2 (for $\mathtt{N} > 0$), whereas `2**(ILEN(N)-1)` rounds `N` down to a power of 2. Compare with `LEADZ`.

The value returned is one *less* than the length of the two's-complement representation of `I`, as the following explains. The shortest two's-complement representation of 4 is 0100. The leading zero is the required sign bit. In 3-bit two's complement, 100 represents -4.

5.6.2 MAXLOC(ARRAY, DIM, MASK)

Optional Arguments. `DIM`, `MASK`

Description. Determine the locations of the first elements of `ARRAY` along dimension `DIM` having the maximum value of the elements identified by `MASK`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer or real. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq \mathtt{DIM} \leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. The result is of type default integer. If `DIM` is absent the result is an array of rank one and size equal to the rank of `ARRAY`; otherwise, the result is an array of rank $n-1$ and shape $(d_1, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$, where $(d_1, \ldots, d_n)$ is the shape of `ARRAY`.

Result Value.

Case (i): The result of executing `S = MAXLOC(ARRAY) + LBOUND(ARRAY) - 1` is a rank-one array `S` of size equal to the rank n of `ARRAY`. It is such that `ARRAY(S(1), ..., S(n))` has the maximum value of all of the elements of `ARRAY`. If more than one element has the maximum value, the element whose subscripts are returned is the first such element, taken in array element order. If `ARRAY` has size zero, the result is implementation dependent.

Case (ii): The result of executing `S = MAXLOC(ARRAY, MASK) + LBOUND(ARRAY) - 1` is a rank-one array `S` of size equal to the rank n of `ARRAY`. It is such that `ARRAY(S(1), ..., S(n))` corresponds to a true element of `MASK`, and has the maximum value of all such elements of `ARRAY`. If more than one element has the maximum value, the element whose subscripts are returned is the first such element, taken in array element order. If there are no such elements (that is, if `ARRAY` has size zero or every element of `MASK` has the value false), the result is implementation dependent.

Case (iii): If `ARRAY` has rank one, the result of `MAXLOC(ARRAY, DIM [,MASK])` is a scalar `S` such that `ARRAY(S + LBOUND(ARRAY,1) - 1)` corresponds to a true element of `MASK` (if `MASK` is present) and has the maximum value of all such elements (all elements if `MASK` is absent). It is the smallest such subscript. Otherwise, the value of element $(s_1, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of
`MAXLOC(ARRAY, DIM [,MASK])` is equal to
`MAXLOC(ARRAY(`$s_1, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`)`
`[,MASK = MASK(`$s_1, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`)])`.

Examples.

Case (i): The value of `MAXLOC((/ 5, -9, 3 /))` is $\begin{bmatrix} 1 \end{bmatrix}$.

Case (ii): `MAXLOC(C, MASK = C .LT. 0)` finds the location of the first element of `C` that is the maximum of the negative elements.

Case (iii): The value of `MAXLOC((/ 5, -9, 3 /), DIM=1)` is 1. If `B` is the array $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$, `MAXLOC( B, DIM = 1 )` is $\begin{bmatrix} 2 & 1 & 2 \end{bmatrix}$ and `MAXLOC( B, DIM = 2 )` is $\begin{bmatrix} 2 & 3 \end{bmatrix}$.
Note that this is true even if `B` has a declared lower bound other than 1.

5.6.3 MINLOC(ARRAY, DIM, MASK)

Optional Arguments. `DIM`, `MASK`

Description. Determine the locations of the first elements of `ARRAY` along dimension `DIM` having the minimum value of the elements identified by `MASK`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer or real. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. The result is of type default integer. If `DIM` is absent the result is an array of rank one and size equal to the rank of `ARRAY`; otherwise, the result is an array of rank $n-1$ and shape $(d_1, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$, where $(d_1, \ldots, d_n)$ is the shape of `ARRAY`.

Result Value.

Case (i): The result of executing `S = MINLOC(ARRAY) + LBOUND(ARRAY) - 1` is a rank-one array `S` of size equal to the rank n of `ARRAY`. It is such that `ARRAY(S(1), ..., S(n))` has the minimum value of all of the elements of `ARRAY`. If more than one element has the minimum value, the element whose subscripts are returned is the first such element, taken in array element order. If `ARRAY` has size zero, the result is implementation dependent.

Case (ii): The result of executing `S = MINLOC(ARRAY, MASK) + LBOUND(ARRAY) - 1` is a rank-one array `S` of size equal to the rank n of `ARRAY`. It is such that `ARRAY(S(1), ..., S(n))` corresponds to a true element of `MASK`, and has the minimum value of all such elements of `ARRAY`. If more than one element has the minimum value, the element whose subscripts are returned is the first such element, taken in array element order. If there are no such elements (that is, if `ARRAY` has size zero or every element of `MASK` has the value false), the result is implementation dependent.

Case (iii): If `ARRAY` has rank one, the result of `MINLOC(ARRAY, DIM [,MASK])` is a scalar `S` such that `ARRAY(S + LBOUND(ARRAY,1) - 1)` corresponds to a true element of `MASK` (if `MASK` is present) and has the minimum value of all such elements (all elements if `MASK` is absent). It is the smallest such subscript. Otherwise, the value of element $(s_1, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of
`MINLOC(ARRAY, DIM [,MASK])` is equal to
`MINLOC( ARRAY(`$(s_1, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n)$`)`
`[,MASK = MASK(`$(s_1, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n)$`)])`.

Examples.

Case (i): The value of `MINLOC((/ 5, -9, 3 /))` is $\left[\ 2\ \right]$.

Case (ii): `MINLOC(C, MASK = C .GT. 0)` finds the location of the first element of `C` that is the minimum of the positive elements.

Case (iii): The value of `MINLOC((/ 5, -9, 3 /), DIM=1)` is 2. If `B` is the array $\begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix}$, `MINLOC( B, DIM = 1 )` is $\begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$ and `MINLOC( B, DIM = 2 )` is $\begin{bmatrix} 3 & 1 \end{bmatrix}$.
Note that this is true even if `B` has a declared lower bound other than 1.

5.6.4 NUMBER_OF_PROCESSORS(DIM)

Optional Argument. `DIM`

Description. Returns the total number of processors available to the program or the number of processors available to the program along a specified dimension of the processor array.

Class. System inquiry function.

Arguments.

`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$ where n is the rank of the processor array.

Result Type, Type Parameter, and Shape. Default integer scalar.

Result Value. The result has a value equal to the extent of dimension `DIM` of the implementation-dependent hardware processor array or, if `DIM` is absent, the total number of elements of the implementation-dependent hardware processor array. The result is always greater than zero.

Examples. For a computer with 8192 processors arranged in a 128 by 64 rectangular grid, the value of `NUMBER_OF_PROCESSORS()` is 8192; the value of `NUMBER_OF_PROCES-SORS(DIM=1)` is 128; and the value of `NUMBER_OF_PROCESSORS(DIM=2)` is 64. For a single-processor workstation, the value of `NUMBER_OF_PROCESSORS()` is 1; since the rank of a scalar processor array is zero, no `DIM` argument may be used.

5.6.5 PROCESSORS_SHAPE()

Description. Returns the shape of the implementation-dependent processor array.

Class. System inquiry function.

Arguments. None

Result Type, Type Parameter, and Shape. The result is a default integer array of rank one whose size is equal to the rank of the implementation-dependent processor array.

Result Value. The value of the result is the shape of the implementation-dependent processor array.

Example. In a computer with 2048 processors arranged in a hypercube, the value of `PROCESSORS_SHAPE()` is [2,2,2,2,2,2,2,2,2,2,2]. In a computer with 8192 processors arranged in a 128 by 64 rectangular grid, the value of `PROCESSORS_SHAPE()` is [128,64]. For a single processor workstation, the value of `PROCESSORS_SHAPE()` is [] (the size-zero array of rank one).

5.7 Specifications of Library Procedures

5.7.1 ALL_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. DIM, SEGMENT, EXCLUSIVE

Description. Computes a segmented logical AND scan along dimension DIM of MASK.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of MASK.
SEGMENT (optional)	must be of type logical and must have the same shape as MASK.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as MASK.

Result Value. Element r of the result has the value ALL((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of MASK selected to contribute to r by the rules stated in Section 5.4.5.

Example. ALL_PREFIX((/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} \mathtt{T} & \mathtt{F} & \mathtt{F} & \mathtt{T} & \mathtt{T} \end{bmatrix}$.

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

Description. Scatters elements of MASK to positions of the result indicated by index arrays INDX1, ..., INDXn. An element of the result is true if and only if the corresponding element of BASE and all elements of MASK scattered to that position are true.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
BASE	must be of type logical with the same kind type parameter as MASK. It must not be scalar.
INDX1,...,INDXn	must be of type integer and conformable with MASK. The number of INDX arguments must be equal to the rank of BASE.

Result Type, Type Parameter, and Shape. Same as BASE.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `ALL( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `MASK` associated with b as described in Section 5.4.4.

Example. `ALL_SCATTER( (/T, T, T, F/), (/T, T, T/), (/1, 1, 2, 2/) )` is $\begin{bmatrix} \mathtt{T} & \mathtt{F} & \mathtt{T} \end{bmatrix}$.

5.7.3 ALL_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented logical AND scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `MASK`.

Result Value. Element r of the result has the value `ALL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `ALL_SUFFIX( (/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} \mathtt{F} & \mathtt{F} & \mathtt{T} & \mathtt{T} & \mathtt{T} \end{bmatrix}$.

5.7.4 ANY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented logical OR scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.

`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `MASK`.

Result Value. Element r of the result has the value `ANY((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `ANY_PREFIX( (/F,T,F,F,F/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} \mathtt{F} & \mathtt{T} & \mathtt{T} & \mathtt{F} & \mathtt{F} \end{bmatrix}$.

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

Description. Scatters elements of `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. An element of the result is true if and only if the corresponding element of `BASE` or any element of `MASK` scattered to that position is true.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`BASE`	must be of type logical with the same kind type parameter as `MASK`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `MASK`. The number of `INDX` arguments must be equal to the rank of `BASE`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `ANY( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `MASK` associated with b as described in Section 5.4.4.

Example. `ANY_SCATTER( (/T, F, F, F/), (/F, F, T/), (/1, 1, 2, 2/) )` is $\begin{bmatrix} \mathtt{T} & \mathtt{F} & \mathtt{T} \end{bmatrix}$.

5.7.6 ANY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented logical OR scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.

DIM (optional)	must be scalar and of type integer with a value in the range $1 \le$ DIM $\le n$, where n is the rank of MASK.
SEGMENT (optional)	must be of type logical and must have the same shape as MASK.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as MASK.

Result Value. Element r of the result has the value ANY((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of MASK selected to contribute to r by the rules stated in Section 5.4.5.

Example. ANY_SUFFIX((/F,T,F,F,F/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} \mathtt{T} & \mathtt{T} & \mathtt{F} & \mathtt{F} & \mathtt{F} \end{bmatrix}$.

5.7.7 COPY_PREFIX(ARRAY, DIM, SEGMENT)

Optional Arguments. DIM, SEGMENT

Description. Computes a segmented copy scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	may be of any type. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \le$ DIM $\le n$, where n is the rank of ARRAY.
SEGMENT (optional)	must be of type logical and must have the same shape as ARRAY.

Result Type, Type Parameter, and Shape. Same as ARRAY.

Result Value. Element r of the result has the value a_1 where $(a_1, \ldots, a_m)$ is the set, in array element order, of elements of ARRAY selected to contribute to r by the rules stated in Section 5.4.5.

Example. COPY_PREFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} 1 & 1 & 1 & 4 & 4 \end{bmatrix}$.

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

Optional Argument. MASK

Description. Scatters elements of ARRAY selected by MASK to positions of the result indicated by index arrays INDX1, ..., INDXn. Each element of the result is equal to one of the elements of ARRAY scattered to that position or, if there is none, to the corresponding element of BASE.

Class. Transformational function.

Arguments.

`ARRAY`	may be of any type. It must not be scalar.
`BASE`	must be of the same type and kind type parameter as `ARRAY`.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of`INDX` arguments must be equal to the rank of `BASE`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. Let S be the set of elements of `ARRAY` associated with element b of `BASE` as described in Secion 5.4.4.

If S is empty, then the element of the result corresponding to the element b of `BASE` has the same value as b.

If S is non-empty, then the element of the result corresponding to the element b of `BASE` is the result of choosing one element from S. HPF does not specify how the choice is to be made; the mechanism is implementation dependent.

Example. `COPY_SCATTER((/1, 2, 3, 4/), (/7, 8, 9/), (/1, 1, 2, 2/))` is [x, y, 9], where x is a member of the set $\{1, 2\}$ and y is a member of the set $\{3, 4\}$.

5.7.9 COPY_SUFFIX(ARRAY, DIM, SEGMENT)

Optional Arguments. `DIM`, `SEGMENT`

Description. Computes a reverse, segmented copy scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	may be of any type. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value a_m where $(a_1, \ldots, a_m)$ is the set, in array element order, of elements of `ARRAY` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `COPY_SUFFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is
$\begin{bmatrix} 3 & 3 & 3 & 5 & 5 \end{bmatrix}$.

5.7.10 COUNT_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented `COUNT` scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. The result is of type default integer and of the same shape as `MASK`.

Result Value. Element r of the result has the value `COUNT((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `COUNT_PREFIX( (/F,T,T,T,T/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 0 & 1 & 2 & 1 & 2 \end{bmatrix}$.

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

Description. Scatters elements of `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. Each element of the result is the sum of the corresponding element of `BASE` and the number of true elements of `MASK` scattered to that position.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`BASE`	must be of type integer. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `MASK`. The number of `INDX` arguments must be equal to the rank of `BASE`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value b `+ COUNT( (/`$a_1, a_2, ..., a_m$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `MASK` associated with b as described in Section 5.4.4.

Example. `COUNT_SCATTER((/T, T, T, F/),(/1, -1, 0/),(/1, 1, 2, 2/))` is $\begin{bmatrix} 3 & 0 & 0 \end{bmatrix}$.

5.7.12 COUNT_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. DIM, SEGMENT, EXCLUSIVE

Description. Computes a reverse, segmented COUNT scan along dimension DIM of MASK.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of MASK.
SEGMENT (optional)	must be of type logical and must have the same shape as MASK.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. The result is of type default integer and of the same shape as MASK.

Result Value. Element r of the result has the value COUNT((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of MASK selected to contribute to r by the rules stated in Section 5.4.5.

Example. COUNT_SUFFIX((/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} 2 & 1 & 1 & 2 & 1 \end{bmatrix}$.

5.7.13 GRADE_DOWN(ARRAY,DIM)

Optional Argument. DIM

Description. Produces a permutation of the indices of an array, sorted by descending array element values.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer, real, or character.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type default integer. If DIM is present, the result has the same shape as ARRAY. If DIM is absent, the result has shape (/ SIZE(SHAPE(ARRAY)), PRODUCT(SHAPE(ARRAY)) /).

Result Value.

Case (i): The result of `S = GRADE_DOWN(ARRAY)` has the property that if one computes the rank-one array `B` of size `PRODUCT(SHAPE(ARRAY))` by `FORALL (K=1:SIZE(B,1)) B(K)=ARRAY(S(1,K),S(2,K),...,S(N,K))` where `N` has the value `SIZE(SHAPE(ARRAY))`, then `B` is sorted in descending order; moreover, all of the columns of `S` are distinct, that is, if $j \neq m$ then $\texttt{ALL(S(:,}j\texttt{) .EQ. S(:,}m\texttt{))}$ will be false. The sort is stable; if $j \leq m$ and $\texttt{B(}j\texttt{)} = \texttt{B(}m\texttt{)}$, then $\texttt{ARRAY(S(}1,j\texttt{),S(}2,j\texttt{),...,S(}n,j\texttt{))}$ precedes $\texttt{ARRAY(S(}1,m\texttt{),S(}2,m\texttt{),...,S(}n,m\texttt{))}$ in the array element ordering of `ARRAY`.

Case (ii): The result of `R = GRADE_DOWN(ARRAY,DIM=K)` has the property that if one computes the array $\texttt{B(}i_1, i_2, \ldots, i_k, \ldots, i_n\texttt{) =}$ $\texttt{ARRAY(}i_1, i_2, \ldots,\ \texttt{R(}i_1, i_2, \ldots, i_k, \ldots, i_n\texttt{)}, \ldots, i_n\ \texttt{)}$ then for all $i_1, i_2, \ldots, (\text{omit } i_k), \ldots, i_n$, the vector $\texttt{B(}i_1, i_2, \ldots, :, \ldots, i_n\texttt{)}$ is sorted in descending order; moreover, $\texttt{R(}i_1, i_2, \ldots, :, \ldots, i_n\texttt{)}$ is a permutation of all the integers in the range `LBOUND(ARRAY,K):UBOUND(ARRAY,K)`. The sort is stable; that is, if $j \leq m$ and $\texttt{B(}i_1, i_2, \ldots, j, \ldots, i_n\texttt{)} = \texttt{B(}i_1, i_2, \ldots, m, \ldots, i_n\texttt{)}$, then $\texttt{R(}i_1, i_2, \ldots, j, \ldots, i_n\texttt{)} \leq \texttt{R(}i_1, i_2, \ldots, m, \ldots, i_n\texttt{)}$.

Examples.

Case (i): `GRADE_DOWN( (/30, 20, 30, 40, -10/) )` is a rank two array of shape $\begin{bmatrix} 1 & 5 \end{bmatrix}$ with the value $\begin{bmatrix} 4 & 1 & 3 & 2 & 5 \end{bmatrix}$. (To produce a rank-one result, the optional `DIM = 1` argument must be used.)

If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `GRADE_DOWN(A)` has the value $\begin{bmatrix} 1 & 2 & 2 & 3 & 3 & 1 & 2 & 1 & 3 \\ 2 & 2 & 1 & 3 & 2 & 3 & 3 & 1 & 1 \end{bmatrix}$.

Case (ii): If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `GRADE_DOWN(A, DIM = 1)` has the value $\begin{bmatrix} 2 & 1 & 3 \\ 1 & 2 & 1 \\ 3 & 3 & 2 \end{bmatrix}$.

5.7.14 GRADE_UP(ARRAY,DIM)

Optional Argument. `DIM`

Description. Produces a permutation of the indices of an array, sorted by ascending array element values.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or character.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \le$ `DIM` $\le n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type default integer. If `DIM` is present, the result has the same shape as `ARRAY`. If `DIM` is absent, the result has shape `(/ SIZE(SHAPE(ARRAY)), PRODUCT(SHAPE(ARRAY)) /)`.

Result Value.

Case (i): The result of `S = GRADE_UP(ARRAY)` has the property that if one computes the rank-one array `B` of size `PRODUCT(SHAPE(ARRAY))` by `FORALL (K=1:SIZE(B,1)) B(K)=ARRAY(S(1,K),S(2,K),...,S(N,K))` where `N` has the value `SIZE(SHAPE(ARRAY))`, then `B` is sorted in ascending order; moreover, all of the columns of `S` are distinct, that is, if $j \neq m$ then `ALL(S(:,`j`) .EQ. S(:,`m`))` will be false. The sort is stable; if $j \le m$ and `B(`j`)` $=$ `B(`m`)`, then `ARRAY(S(1,`j`),S(2,`j`),...,S(`n,j`))` precedes `ARRAY(S(1,`m`),S(2,`m`),...,S(`n,m`))` in the array element ordering of `ARRAY`.

Case (ii): The result of `R = GRADE_UP(ARRAY,DIM=K)` has the property that if one computes the array `B(`$i_1, i_2, \ldots, i_k, \ldots, i_n$`) =` `ARRAY(`$i_1, i_2, \ldots,$ `R(`$i_1, i_2, \ldots, i_k, \ldots, i_n$`)`$, \ldots, i_n$ `)` then for all $i_1, i_2, \ldots,$ (omit i_k)$, \ldots, i_n$, the vector `B(`$i_1, i_2, \ldots, :, \ldots, i_n$`)` is sorted in ascending order; moreover, `R(`$i_1, i_2, \ldots, :, \ldots, i_n$`)` is a permutation of all the integers in the range `LBOUND(ARRAY,K):UBOUND(ARRAY,K)`. The sort is stable; that is, if $j \le m$ and `B(`$i_1, i_2, \ldots, j, \ldots, i_n$`)` $=$ `B(`$i_1, i_2, \ldots, m, \ldots, i_n$`)`, then `R(`$i_1, i_2, \ldots, j, \ldots, i_n$`)` $\le$ `R(`$i_1, i_2, \ldots, m, \ldots, i_n$`)`.

Examples.

Case (i): `GRADE_UP( (/30, 20, 30, 40, -10/) )` is a rank two array of shape $\begin{bmatrix} 1 & 5 \end{bmatrix}$ with the value $\begin{bmatrix} 5 & 2 & 1 & 3 & 4 \end{bmatrix}$. (To produce a rank-one result, the optional `DIM = 1` argument must be used.)

If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `GRADE_UP(A)` has the value $\begin{bmatrix} 1 & 3 & 3 & 1 & 2 & 2 & 3 & 2 & 1 \\ 1 & 1 & 2 & 3 & 3 & 1 & 3 & 2 & 2 \end{bmatrix}$.

Case (ii): If `A` is the array $\begin{bmatrix} 1 & 9 & 2 \\ 4 & 5 & 2 \\ 1 & 2 & 4 \end{bmatrix}$,

then `GRADE_UP(A, DIM = 1)` has the value $\begin{bmatrix} 1 & 3 & 1 \\ 3 & 2 & 2 \\ 2 & 1 & 3 \end{bmatrix}$.

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

Optional Arguments. LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES

Description. Returns information regarding the correspondence of a variable and the *align-target* (array or template) to which it is ultimately aligned.

Class. Mapping inquiry subroutine.

Arguments.

ALIGNEE — may be of any type. It may be scalar or array valued. It must not be an assumed-size array. It must not be a structure component. If it is a member of an aggregate variable group, then it must be an aggregate cover of the group. (See Section 7 for the definitions of "aggregate variable group" and "aggregate cover.") It must not be a pointer that is disassociated or an allocatable array that is not allocated. It is an INTENT (IN) argument.

If ALIGNEE is a pointer, information about the alignment of its target is returned. The target must not be an assumed-size dummy argument or a section of an assumed-size dummy argument. If the target is (a section of) a member of an aggregate variable group, then the member must be an aggregate cover of the group. The target must not be a structure component, but the pointer may be.

LB (optional) — must be of type default integer and of rank one. Its size must be at least equal to the rank of ALIGNEE. It is an INTENT (OUT) argument. The first element of the i^{th} axis of ALIGNEE is ultimately aligned to the LB(i)th *align-target* element along the axis of the *align-target* associated with the i^{th} axis of ALIGNEE. If the i^{th} axis of ALIGNEE is a collapsed axis, LB(i) is implementation dependent.

UB (optional) — must be of type default integer and of rank one. Its size must be at least equal to the rank of ALIGNEE. It is an INTENT (OUT) argument. The last element of the i^{th} axis of ALIGNEE is ultimately aligned to the UB(i)th *align-target* element along the axis of the *align-target* associated with the i^{th} axis of ALIGNEE. If the i^{th} axis of ALIGNEE is a collapsed axis, UB(i) is implementation dependent.

STRIDE (optional) — must be of type default integer and of rank one. Its size must be at least equal to the rank of ALIGNEE. It is an INTENT (OUT) argument. The i^{th} element of STRIDE is set to the stride used in aligning the elements of ALIGNEE along its i^{th} axis. If the i^{th} axis of ALIGNEE is a collapsed axis, STRIDE(i) is zero.

AXIS_MAP (optional) must be of type default integer and of rank one. Its size must be at least equal to the rank of ALIGNEE. It is an INTENT (OUT) argument. The i^{th} element of AXIS_MAP is set to the *align-target* axis associated with the i^{th} axis of ALIGNEE. If the i^{th} axis of ALIGNEE is a collapsed axis, AXIS_MAP(i) is 0.

IDENTITY_MAP (optional) must be scalar and of type default logical. It is an INTENT (OUT) argument. It is set to true if the ultimate *align-target* associated with ALIGNEE has a shape identical to ALIGNEE, the axes are mapped using the identity permutation, and the strides are all positive (and therefore equal to 1, because of the shape constraint); otherwise it is set to false. If a variable has not appeared as an *alignee* in an ALIGN or REALIGN directive, and does not have the INHERIT attribute, then IDENTITY_MAP must be true; it can be true in other circumstances as well.

DYNAMIC (optional) must be scalar and of type default logical. It is an INTENT (OUT) argument. It is set to true if ALIGNEE has the DYNAMIC attribute; otherwise it is set to false. If ALIGNEE has the pointer attribute, then the result applies to ALIGNEE itself rather than its target.

NCOPIES (optional) must be scalar and of type default integer. It is an INTENT (OUT) argument. It is set to the number of copies of ALIGNEE that are ultimately aligned to *align-target*. For a non-replicated variable, it is set to one.

Examples. If ALIGNEE is scalar, then no elements of LB, UB, STRIDE, or AXIS_MAP are set.

Given the declarations

```
      REAL PI = 3.1415927
      POINTER P_TO_A(:)
      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
      P_TO_A => A(3:9:2, 6)
```

assuming that the actual mappings are as the directives specify, the results of HPF_ALIGNMENT are:

	A	B	C	D	P_TO_A
LB	[4, 2]	[1, 1]	[20, N/A, 1]	[1]	[10]
UB	[31, 20]	[20, 30]	[1, N/A, 10]	[40]	[28]
STRIDE	[3, 2]	[1, 1]	[-1, 0, 1]	[1]	[6]
AXIS_MAP	[1, 2]	[1, 2]	[2, 0, 1]	[1]	[1]
IDENTITY_MAP	false	true	false	false	false
DYNAMIC	true	false	false	false	false
NCOPIES	1	1	1	1	1

where "N/A" denotes a implementation-dependent result. To illustrate the use of `NCOPIES`, consider:

```
      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)
```

`CALL HPF_ALIGNMENT(RONALD_MCDONALD, NCOPIES = NC)` sets `NC` to 20. Now consider:

```
      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)` sets `NC` to one.

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

Optional Arguments. `LB`, `UB`, `AXIS_TYPE`, `AXIS_INFO`, `NUMBER_ALIGNED`, `TEMPLATE_RANK`, `DYNAMIC`

Description. The `HPF_TEMPLATE` subroutine returns information regarding the ultimate *align-target* associated with a variable; `HPF_TEMPLATE` returns information concerning the variable from the template's point of view (assuming the alignment is to a template rather than to an array), while `HPF_ALIGNMENT` returns information from the variable's point of view.

Class. Mapping inquiry subroutine.

Arguments.

`ALIGNEE` may be of any type. It may be scalar or array valued. It must not be an assumed-size array. It must not be a structure component. If it is a member of an aggregate variable group, then it must be an aggregate cover of the group. (See Section 7 for the definitions of "aggregate variable group" and "aggregate cover.") It must not be a pointer that is disassociated or an allocatable array that is not allocated. It is an `INTENT (IN)` argument.

If `ALIGNEE` is a pointer, information about the alignment of its target is returned. The target must not be an assumed-size dummy argument or a section of an assumed-size dummy argument. If the target is (a section of) a member of an aggregate variable group, then the member must be an aggregate cover of the group. The target must not be a structure component, but the pointer may be.

`TEMPLATE_RANK` (optional) must be scalar and of type default integer. It is an `INTENT (OUT)` argument. It is set to the rank of the ultimate *align-target*. This can be different from the rank of the `ALIGNEE`, due to collapsing and replicating.

`LB` (optional) must be of type default integer and of rank one. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. The i^{th} element of `LB` contains the declared *align-target* lower bound for the i^{th} template axis.

`UB` (optional) must be of type default integer and of rank one. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. The i^{th} element of `UB` contains the declared *align-target* upper bound for the i^{th} template axis.

`AXIS_TYPE` (optional) must be a rank one array of type default character. It may be of any length, although it must be of length at least 10 in order to contain the complete value. Its elements are set to the values below as if by a character intrinsic assignment statement. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. The i^{th} element of `AXIS_TYPE` contains information about the i^{th} axis of the *align-target*. The following values are defined by HPF (implementations may define other values):

`'NORMAL'` The *align-target* axis has an axis of `ALIGNEE` aligned to it. For elements of `AXIS_TYPE` assigned this value, the corresponding element of `AXIS_INFO` is set to the number of the axis of `ALIGNEE` aligned to this *align-target* axis.

`'REPLICATED'` `ALIGNEE` is replicated along this *align-target* axis. For elements of `AXIS_TYPE` assigned this value, the corresponding element of `AXIS_INFO` is set to the number of copies of `ALIGNEE` along this *align-target* axis.

`'SINGLE'` `ALIGNEE` is aligned with one coordinate of the *align-target* axis. For elements of `AXIS_TYPE` assigned this value, the corresponding element of `AXIS_INFO` is set to the *align-target* coordinate to which `ALIGNEE` is aligned.

`AXIS_INFO` (optional) must be of type default integer and of rank one. Its size must be at least equal to the rank of the *align-target* to which `ALIGNEE` is ultimately aligned; this is the value returned in `TEMPLATE_RANK`. It is an `INTENT (OUT)` argument. See the description of `AXIS_TYPE` above.

`NUMBER_ALIGNED` (optional) must be scalar and of type default integer. It is an `INTENT (OUT)` argument. It is set to the total number of variables aligned to the ultimate *align-target*. This is the number of variables that are moved if the *align-target* is redistributed.

`DYNAMIC` (optional) must be scalar and of type default logical. It is an `INTENT (OUT)` argument. It is set to true if the *align-target* has the `DYNAMIC` attribute, and to false otherwise.

Example. Given the declarations in the example of Section 5.7.15, and assuming that the actual mappings are as the directives specify, the results of `HPF_TEMPLATE` are:

	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

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

Optional Arguments. `AXIS_TYPE`, `AXIS_INFO`, `PROCESSORS_RANK`, `PROCESSORS_SHAPE`

Description. The `HPF_DISTRIBUTION` subroutine returns information regarding the distribution of the ultimate *align-target* associated with a variable.

Class. Mapping inquiry subroutine.

Arguments.

`DISTRIBUTEE` may be of any type. It may be scalar or array valued. It must not be an assumed-size array. It must not be a structure component. If it is a member of an aggregate variable group, then it must be an aggregate cover of the group. (See Section 7 for the definitions of "aggregate variable group" and "aggregate cover.") It must not be a pointer that is disassociated or an allocatable array that is not allocated. It is an `INTENT (IN)` argument.

If `DISTRIBUTEE` is a pointer, information about the distribution of its target is returned. The target must not be an assumed-size dummy argument or a section of an assumed-size dummy argument. If the target is (a section of) a member of an aggregate variable group, then the member must be an aggregate cover of the group. The target must not be a structure component, but the pointer may be.

`AXIS_TYPE` (optional) must be a rank one array of type default character. It may be of any length, although it must be of length at least 9 in order to contain the complete value. Its elements are set to the values below as if by a character intrinsic assignment statement. Its size must be at least equal to the rank of the *align-target* to which `DISTRIBUTEE` is ultimately aligned; this is the value returned by `HPF_TEMPLATE` in `TEMPLATE_RANK`). It is an `INTENT (OUT)` argument. Its i^{th} element contains information on the distribution of the i^{th} axis of that *align-target*. The following values are defined by HPF (implementations may define other values):

`'BLOCK'` The axis is distributed `BLOCK`. The corresponding element of `AXIS_INFO` contains the block size.

`'COLLAPSED'` The axis is collapsed (distributed with the "*" specification). The value of the corresponding element of `AXIS_INFO` is implementation dependent.

`'CYCLIC'` The axis is distributed `CYCLIC`. The corresponding element of `AXIS_INFO` contains the block size.

`AXIS_INFO` (optional) must be a rank one array of type default integer, and size at least equal to the rank of the *align-target* to which `DISTRIBUTEE` is ultimately aligned (which is returned by `HPF_TEMPLATE` in `TEMPLATE_RANK`). It is an `INTENT (OUT)` argument. The i^{th} element of `AXIS_INFO` contains the block size in the block or cyclic distribution of the i^{th} axis of the ultimate *align-target* of `DISTRIBUTEE`; if that axis is a collapsed axis, then the value is implementation dependent.

`PROCESSORS_RANK` (optional) must be scalar and of type default integer. It is set to the rank of the processor arrangement onto which `DISTRIBUTEE` is distributed. It is an `INTENT (OUT)` argument.

`PROCESSORS_SHAPE` (optional) must be a rank one array of type default integer and of size at least equal to the value, m, returned in `PROCESSORS_RANK`. It is an `INTENT (OUT)` argument. Its first m

elements are set to the shape of the processor arrangement onto which `DISTRIBUTEE` is mapped. (It may be necessary to call `HPF_DISTRIBUTION` twice, the first time to obtain the value of `PROCESSORS_RANK` in order to allocate `PROCESSORS_SHAPE`.)

Example. Given the declarations in the example of Section 5.7.15, and assuming that the actual mappings are as the directives specify, the results of `HPF_DISTRIBUTION` are:

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

5.7.18 IALL(ARRAY, DIM, MASK)

Optional Arguments. `DIM`, `MASK`

Description. Computes a bitwise logical AND reduction along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY` must be of type integer. It must not be scalar.

`DIM` (optional) must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.

`MASK` (optional) must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. The result is of type integer with the same kind type parameter as `ARRAY`. It is scalar if `DIM` is absent or if `ARRAY` has rank one; otherwise, the result is an array of rank $n-1$ and shape $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of `ARRAY`.

Result Value.

Case (i): The result of `IALL(ARRAY)` is the `IAND` reduction of all the elements of `ARRAY`. If `ARRAY` has size zero, the result is equal to a implementation-dependent integer value x with the property that `IAND(I, x) = I` for all integers `I` of the same kind type parameter as `ARRAY`. See Section 5.4.3.

Case (ii): The result of `IALL(ARRAY, MASK=MASK)` is the `IAND` reduction of all the elements of `ARRAY` corresponding to the true elements of `MASK`; if `MASK` contains no true elements, the result is equal to a implementation-dependent integer value x (of the same kind type parameter as `ARRAY`) with the property that `IAND(I, x) = I` for all integers `I`.

Case (iii): If `ARRAY` has rank one, `IALL(ARRAY, DIM [,MASK])` has a value equal to that of `IALL(ARRAY [,MASK])`. Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of `IALL(ARRAY, DIM [,MASK])` is equal to `IALL(ARRAY(`$s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`)` `[,MASK = MASK(`$s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`)])`

Examples.

Case (i): The value of `IALL((/7, 6, 3, 2/))` is 2.

Case (ii): The value of `IALL(C, MASK = BTEST(C,0))` is the `IAND` reduction of the odd elements of `C`.

Case (iii): If `B` is the array $\begin{bmatrix} 2 & 3 & 5 \\ 3 & 7 & 7 \end{bmatrix}$, then `IALL(B, DIM = 1)` is $\begin{bmatrix} 2 & 3 & 5 \end{bmatrix}$ and `IALL(B, DIM = 2)` is $\begin{bmatrix} 0 & 3 \end{bmatrix}$.

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

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented bitwise logical AND scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `IALL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `IALL_PREFIX( (/1,3,2,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 1 & 0 & 4 & 4 \end{bmatrix}$.

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

Optional Argument. MASK

Description. Scatters elements of ARRAY selected by MASK to positions of the result indicated by index arrays INDX1, ..., INDXn. The j^{th}bit of an element of the result is 1 if and only if the j^{th}bits of the corresponding element of BASE and of the elements of ARRAY scattered to that position are all equal to 1.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer. It must not be scalar.
BASE	must be of type integer with the same kind type parameter as ARRAY. It must not be scalar.
INDX1,...,INDXn	must be of type integer and conformable with ARRAY. The number of INDX arguments must be equal to the rank of BASE.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. Same as BASE.

Result Value. The element of the result corresponding to the element b of BASE has the value IALL((/$a_1, a_2, ..., a_m, b$/)), where $(a_1, \ldots, a_m)$ are the elements of ARRAY associated with b as described in Section 5.4.4.

Example. IALL_SCATTER((/1, 2, 3, 6/), (/1, 3, 7/), (/1, 1, 2, 2/)) is $\begin{bmatrix} 0 & 2 & 7 \end{bmatrix}$.

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

Optional Arguments. DIM, MASK, SEGMENT, EXCLUSIVE

Description. Computes a reverse, segmented bitwise logical AND scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.
SEGMENT (optional)	must be of type logical and must have the same shape as ARRAY.

`EXCLUSIVE` (optional) must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `IALL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `IALL_SUFFIX( (/1,3,2,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 0 & 2 & 2 & 4 & 5 \end{bmatrix}$.

5.7.22 IANY(ARRAY, DIM, MASK)

Optional Arguments. `DIM`, `MASK`

Description. Computes a bitwise logical OR reduction along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. The result is of type integer with the same kind type parameter as `ARRAY`. It is scalar if `DIM` is absent or if `ARRAY` has rank one; otherwise, the result is an array of rank $n-1$ and shape $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of `ARRAY`.

Result Value.

Case (i): The result of `IANY(ARRAY)` is the `IOR` reduction of all the elements of `ARRAY`. If `ARRAY` has size zero, the result has the value zero. See Section 5.4.3.

Case (ii): The result of `IANY(ARRAY, MASK=MASK)` is the `IOR` reduction of all the elements of `ARRAY` corresponding to the true elements of `MASK`; if `MASK` contains no true elements, the result is zero.

Case (iii): If `ARRAY` has rank one, `IANY(ARRAY, DIM [,MASK])` has a value equal to that of `IANY(ARRAY [,MASK])`. Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of `IANY(ARRAY, DIM [,MASK])` is equal to `IANY(ARRAY(`$s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`)` `[,MASK = MASK(`$s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`)])`

Examples.

Case (i): The value of `IANY((/9, 8, 3, 2/))` is 11.

Case (ii): The value of `IANY(C, MASK = BTEST(C,0))` is the `IOR` reduction of the odd elements of `C`.

Case (iii): If `B` is the array $\begin{bmatrix} 2 & 3 & 5 \\ 0 & 4 & 2 \end{bmatrix}$, then `IANY(B, DIM = 1)` is $\begin{bmatrix} 2 & 7 & 7 \end{bmatrix}$ and `IANY(B, DIM = 2)` is $\begin{bmatrix} 7 & 6 \end{bmatrix}$.

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

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented bitwise logical OR scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `IANY((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `IANY_PREFIX( (/1,2,3,2,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 3 & 3 & 2 & 7 \end{bmatrix}$.

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

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1, ..., INDXn`. The j$^{\text{th}}$bit of an element of the result is 1 if and only if the j$^{\text{th}}$bit of the corresponding element of `BASE` or of any of the elements of `ARRAY` scattered to that position is equal to 1.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer. It must not be scalar.
BASE	must be of type integer with the same kind type parameter as ARRAY. It must not be scalar.
INDX1,...,INDXn	must be of type integer and conformable with ARRAY. The number of INDX arguments must be equal to the rank of BASE.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. Same as BASE.

Result Value. The element of the result corresponding to the element b of BASE has the value IANY((/$a_1, a_2, ..., a_m, b$/)), where $(a_1, \ldots, a_m)$ are the elements of ARRAY associated with b as described in Section 5.4.4.

Example. IANY_SCATTER((/1, 2, 3, 6/), (/1, 3, 7/), (/1, 1, 2, 2/)) is $\begin{bmatrix} 3 & 7 & 7 \end{bmatrix}$.

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

Optional Arguments. DIM, MASK, SEGMENT, EXCLUSIVE

Description. Computes a reverse, segmented bitwise logical OR scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.
SEGMENT (optional)	must be of type logical and must have the same shape as ARRAY.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as ARRAY.

Result Value. Element r of the result has the value IANY((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of ARRAY selected to contribute to r by the rules stated in Section 5.4.5.

Example. IANY_SUFFIX((/4,2,3,2,5/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} 7 & 3 & 3 & 7 & 5 \end{bmatrix}$.

5.7.26 IPARITY(ARRAY, DIM, MASK)

Optional Arguments. DIM, MASK

Description. Computes a bitwise logical exclusive OR reduction along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. The result is of type integer with the same kind type parameter as ARRAY. It is scalar if DIM is absent or if ARRAY has rank one; otherwise, the result is an array of rank $n-1$ and shape $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of ARRAY.

Result Value.

Case (i): The result of IPARITY(ARRAY) is the IEOR reduction of all the elements of ARRAY. If ARRAY has size zero, the result has the value zero. See Section 5.4.3.

Case (ii): The result of IPARITY(ARRAY, MASK=MASK) is the IEOR reduction of all the elements of ARRAY corresponding to the true elements of MASK; if MASK contains no true elements, the result is zero.

Case (iii): If ARRAY has rank one, IPARITY(ARRAY, DIM [,MASK]) has a value equal to that of IPARITY(ARRAY [,MASK]). Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of IPARITY(ARRAY, DIM [,MASK]) is equal to IPARITY(ARRAY($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$) [,MASK = MASK($s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$)])

Examples.

Case (i): The value of IPARITY((/13, 8, 3, 2/)) is 4.

Case (ii): The value of IPARITY(C, MASK = BTEST(C,0)) is the IEOR reduction of the odd elements of C.

Case (iii): If B is the array $\begin{bmatrix} 2 & 3 & 7 \\ 0 & 4 & 2 \end{bmatrix}$, then IPARITY(B, DIM = 1) is $\begin{bmatrix} 2 & 7 & 5 \end{bmatrix}$ and IPARITY(B, DIM = 2) is $\begin{bmatrix} 6 & 6 \end{bmatrix}$.

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

Optional Arguments. DIM, MASK, SEGMENT, EXCLUSIVE

Description. Computes a segmented bitwise logical exclusive OR scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.
SEGMENT (optional)	must be of type logical and must have the same shape as ARRAY.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as ARRAY.

Result Value. Element r of the result has the value IPARITY((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of ARRAY selected to contribute to r by the rules stated in Section 5.4.5.

Example. IPARITY_PREFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} 1 & 3 & 0 & 4 & 1 \end{bmatrix}$.

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

Optional Argument. MASK

Description. Scatters elements of ARRAY selected by MASK to positions of the result indicated by index arrays INDX1, ..., INDXn. The j^{th}bit of an element of the result is 1 if and only if there are an odd number of ones among the j^{th}bits of the corresponding element of BASE and the elements of ARRAY scattered to that position.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer. It must not be scalar.
BASE	must be of type integer with the same kind type parameter as ARRAY. It must not be scalar.
INDX1,...,INDXn	must be of type integer and conformable with ARRAY. The number of INDX arguments must be equal to the rank of BASE.

MASK (optional) must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. Same as BASE.

Result Value. The element of the result corresponding to the element b of BASE has the value IPARITY((/$a_1, a_2, ..., a_m, b$/)), where $(a_1, \ldots, a_m)$ are the elements of ARRAY associated with b as described in Section 5.4.4.

Example. IPARITY_SCATTER((/1,2,3,6/), (/1,3,7/), (/1,1,2,2/)) is $\begin{bmatrix} 2 & 6 & 7 \end{bmatrix}$.

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

Optional Arguments. DIM, MASK, SEGMENT, EXCLUSIVE

Description. Computes a reverse, segmented bitwise logical exclusive OR scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.
SEGMENT (optional)	must be of type logical and must have the same shape as ARRAY.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as ARRAY.

Result Value. Element r of the result has the value IPARITY((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of ARRAY selected to contribute to r by the rules stated in Section 5.4.5.

Example. IPARITY_SUFFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} 0 & 1 & 3 & 1 & 5 \end{bmatrix}$.

5.7.30 LEADZ(I)

Description. Return the number of leading zeros in an integer.

Class. Elemental function.

Argument. I must be of type integer.

Result Type and Type Parameter. Same as I.

Result Value. The result is a count of the number of leading 0-bits in the integer I. The model for the interpretation of an integer as a sequence of bits is in Section 13.5.7 of the Fortran 90 Standard. LEADZ(0) is BIT_SIZE(I). For nonzero I, if the leftmost one bit of I occurs in position $k-1$ (where the rightmost bit is bit 0) then LEADZ(I) is BIT_SIZE(I) - k.

Examples. LEADZ(3) has the value BIT_SIZE(3) - 2. For scalar I, LEADZ(I) .EQ. MINVAL((/ (J, J=0, BIT_SIZE(I)) /), MASK=M) where M =(/ (BTEST(I,J), J=BIT_SIZE(I)-1, 0, -1), .TRUE. /). A given integer I may produce different results from LEADZ(I), depending on the number of bits in the representation of the integer (BIT_SIZE(I)). That is because LEADZ counts bits from the most significant bit. Compare with ILEN.

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

Optional Arguments. DIM, MASK, SEGMENT, EXCLUSIVE

Description. Computes a segmented MAXVAL scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.
SEGMENT (optional)	must be of type logical and must have the same shape as ARRAY.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as ARRAY.

Result Value. Element r of the result has the value MAXVAL((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of ARRAY selected to contribute to r by the rules stated in Section 5.4.5.

Example. MAXVAL_PREFIX((/3,4,-5,2,5/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} 3 & 4 & 4 & 2 & 5 \end{bmatrix}$.

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

Optional Argument. MASK

Description. Scatters elements of ARRAY selected by MASK to positions of the result indicated by index arrays INDX1, ..., INDXn. Each element of the result is assigned the maximum value of the corresponding element of BASE and the elements of ARRAY scattered to that position.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
BASE	must be of the same type and kind type parameter as ARRAY. It must not be scalar.
INDX1,...,INDXn	must be of type integer and conformable with ARRAY. The number of INDX arguments must be equal to the rank of BASE.
MASK (optional)	must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. Same as BASE.

Result Value. The element of the result corresponding to the element b of BASE has the value MAXVAL((/$a_1, a_2, ..., a_m, b$/)), where $(a_1, \ldots, a_m)$ are the elements of ARRAY associated with b as described in Section 5.4.4.

Example. MAXVAL_SCATTER((/1, 2, 3, 1/), (/4, -5, 7/), (/1, 1, 2, 2/)) is $\begin{bmatrix} 4 & 3 & 7 \end{bmatrix}$.

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

Optional Arguments. DIM, MASK, SEGMENT, EXCLUSIVE

Description. Computes a reverse, segmented MAXVAL scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.
SEGMENT (optional)	must be of type logical and must have the same shape as ARRAY.

EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as ARRAY.

Result Value. Element r of the result has the value MAXVAL((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of ARRAY selected to contribute to r by the rules stated in Section 5.4.5.

Example. MAXVAL_SUFFIX((/3,4,-5,2,5/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} 4 & 4 & -5 & 5 & 5 \end{bmatrix}$.

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

Optional Arguments. DIM, MASK, SEGMENT, EXCLUSIVE

Description. Computes a segmented MINVAL scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer or real. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.
SEGMENT (optional)	must be of type logical and must have the same shape as ARRAY.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as ARRAY.

Result Value. Element r of the result has the value MINVAL((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of ARRAY selected to contribute to r by the rules stated in Section 5.4.5.

Example. MINVAL_PREFIX((/1,2,-3,4,5/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} 1 & 1 & -3 & 4 & 4 \end{bmatrix}$.

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

Optional Argument. MASK

Description. Scatters elements of ARRAY selected by MASK to positions of the result indicated by index arrays INDX1, ..., INDXn. Each element of the result is assigned the minimum value of the corresponding element of BASE and the elements of ARRAY scattered to that position.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer or real. It must not be scalar.
`BASE`	must be of the same type and kind type parameter as `ARRAY`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of `INDX` arguments must be equal to the rank of `BASE`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `MINVAL(` `(/`$a_1, a_2, ..., a_m, b$`/)` `)`, where $(a_1, \ldots, a_m)$ are the elements of `ARRAY` associated with b as described in Section 5.4.4.

Example. `MINVAL_SCATTER((/ 1,-2,-3,6 /), (/ 4,3,7 /), (/ 1,1,2,2 /))` is $\begin{bmatrix} -2 & -3 & 7 \end{bmatrix}$.

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

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented `MINVAL` scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer or real. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `MINVAL((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `MINVAL_SUFFIX( (/1,2,-3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} -3 & -3 & -3 & 4 & 5 \end{bmatrix}$.

5.7.37 PARITY(MASK, DIM)

Optional Argument. `DIM`

Description. Determine whether an odd number of values are true in `MASK` along dimension `DIM`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter, and Shape. The result is of type logical with the same kind type parameter as `MASK`. It is scalar if `DIM` is absent or if `MASK` has rank one; otherwise, the result is an array of rank $n-1$ and shape $(d_1, d_2, \ldots, d_{DIM-1}, d_{DIM+1}, \ldots, d_n)$ where $(d_1, d_2, \ldots, d_n)$ is the shape of `MASK`.

Result Value.

Case (i): The result of `PARITY(MASK)` is the `.NEQV.` reduction of all the elements of `MASK`. If `MASK` has size zero, the result has the value false. See Section 5.4.3.

Case (ii): If `MASK` has rank one, `PARITY(MASK, DIM)` has a value equal to that of `PARITY(MASK)`. Otherwise, the value of element $(s_1, s_2, \ldots, s_{DIM-1}, s_{DIM+1}, \ldots, s_n)$ of `PARITY(MASK, DIM)` is equal to `PARITY(MASK(`$s_1, s_2, \ldots, s_{DIM-1}, :, s_{DIM+1}, \ldots, s_n$`))`

Examples.

Case (i): The value of `PARITY((/T, T, T, F/))` is true.

Case (ii): If `B` is the array $\begin{bmatrix} \mathtt{T} & \mathtt{T} & \mathtt{F} \\ \mathtt{T} & \mathtt{T} & \mathtt{T} \end{bmatrix}$, then `PARITY(B, DIM = 1)` is $\begin{bmatrix} \mathtt{F} & \mathtt{F} & \mathtt{T} \end{bmatrix}$ and `PARITY(B, DIM = 2)` is $\begin{bmatrix} \mathtt{F} & \mathtt{T} \end{bmatrix}$.

5.7.38 PARITY_PREFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented logical exclusive OR scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.

`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `MASK`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `MASK`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `MASK`.

Result Value. Element r of the result has the value `PARITY((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `MASK` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `PARITY_PREFIX( (/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} \mathtt{T} & \mathtt{T} & \mathtt{F} & \mathtt{T} & \mathtt{F} \end{bmatrix}$.

5.7.39 PARITY_SCATTER(MASK,BASE,INDX1, ..., INDXn)

Description. Scatters elements of `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. An element of the result is true if and only if the number of true values among the corresponding element of `BASE` and the elements of `MASK` scattered to that position is odd.

Class. Transformational function.

Arguments.

`MASK`	must be of type logical. It must not be scalar.
`BASE`	must be of type logical with the same kind type parameter as `MASK`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `MASK`. The number of `INDX` arguments must be equal to the rank of `BASE`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `PARITY( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `MASK` associated with b as described in Section 5.4.4.

Example. `PARITY_SCATTER((/ T,T,T,T /), (/ T,F,F /), (/ 1,1,1,2 /))` is $\begin{bmatrix} \mathtt{F} & \mathtt{T} & \mathtt{F} \end{bmatrix}$.

5.7.40 PARITY_SUFFIX(MASK, DIM, SEGMENT, EXCLUSIVE)

Optional Arguments. `DIM`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a reverse, segmented logical exclusive OR scan along dimension `DIM` of `MASK`.

Class. Transformational function.

Arguments.

MASK	must be of type logical. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of MASK.
SEGMENT (optional)	must be of type logical and must have the same shape as MASK.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as MASK.

Result Value. Element r of the result has the value PARITY((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of MASK selected to contribute to r by the rules stated in Section 5.4.5.

Example. `PARITY_SUFFIX( (/T,F,T,T,T/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} \mathtt{F} & \mathtt{T} & \mathtt{T} & \mathtt{F} & \mathtt{T} \end{bmatrix}$.

5.7.41 POPCNT(I)

Description. Return the number of one bits in an integer.

Class. Elemental function.

Argument. I must be of type integer.

Result Type and Type Parameter. Same as I.

Result Value. POPCNT(I) is the number of one bits in the binary representation of the integer I. The model for the interpretation of an integer as a sequence of bits is in Section 13.5.7 of the Fortran 90 Standard.

Example. `POPCNT(I) = COUNT((/ (BTEST(I,J), J=0, BIT_SIZE(I)-1) /))`, for scalar I.

5.7.42 POPPAR(I)

Description. Return the parity of an integer.

Class. Elemental function.

Argument. I must be of type integer.

Result Type and Type Parameter. Same as I.

Result Value. POPPAR(I) is 1 if there are an odd number of one bits in I and zero if there are an even number. The model for the interpretation of an integer as a sequence of bits is in Section 13.5.7 of the Fortran 90 Standard.

Example. For scalar I, `POPPAR(I) = MERGE(1,0,BTEST(POPCNT(I),0))`.

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

Optional Arguments. `DIM`, `MASK`, `SEGMENT`, `EXCLUSIVE`

Description. Computes a segmented `PRODUCT` scan along dimension `DIM` of `ARRAY`.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `PRODUCT((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `PRODUCT_PREFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 2 & 6 & 4 & 20 \end{bmatrix}$.

5.7.44 PRODUCT_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. Each element of the result is equal to the product of the corresponding element of `BASE` and the elements of `ARRAY` scattered to that position.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`BASE`	must be of the same type and kind type parameter as `ARRAY`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of `INDX` arguments must be equal to the rank of `BASE`.

MASK (optional) — must be of type logical and must be conformable with ARRAY.

Result Type, Type Parameter, and Shape. Same as BASE.

Result Value. The element of the result corresponding to the element b of BASE has the value PRODUCT((/$a_1, a_2, ..., a_m, b$/)), where $(a_1, \ldots, a_m)$ are the elements of ARRAY associated with b as described in Section 5.4.4.

Example. PRODUCT_SCATTER((/ 1,2,3,1 /), (/ 4,-5,7 /), (/ 1,1,2,2 /)) is $\begin{bmatrix} 8 & -15 & 7 \end{bmatrix}$.

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

Optional Arguments. DIM, MASK, SEGMENT, EXCLUSIVE

Description. Computes a reverse, segmented PRODUCT scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY — must be of type integer, real, or complex. It must not be scalar.

DIM (optional) — must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.

MASK (optional) — must be of type logical and must be conformable with ARRAY.

SEGMENT (optional) — must be of type logical and must have the same shape as ARRAY.

EXCLUSIVE (optional) — must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as ARRAY.

Result Value. Element r of the result has the value PRODUCT((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of ARRAY selected to contribute to r by the rules stated in Section 5.4.5.

Example. PRODUCT_SUFFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} 6 & 6 & 3 & 20 & 5 \end{bmatrix}$.

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

Optional Arguments. DIM, MASK, SEGMENT, EXCLUSIVE

Description. Computes a segmented SUM scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \le$ `DIM` $\le n$, where n is the rank of `ARRAY`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.
`SEGMENT` (optional)	must be of type logical and must have the same shape as `ARRAY`.
`EXCLUSIVE` (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as `ARRAY`.

Result Value. Element r of the result has the value `SUM((/` $a_1, \ldots, a_m$ `/))` where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of `ARRAY` selected to contribute to r by the rules stated in Section 5.4.5.

Example. `SUM_PREFIX( (/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/) )` is $\begin{bmatrix} 1 & 3 & 6 & 4 & 9 \end{bmatrix}$.

5.7.47 SUM_SCATTER(ARRAY,BASE,INDX1, ..., INDXn, MASK)

Optional Argument. `MASK`

Description. Scatters elements of `ARRAY` selected by `MASK` to positions of the result indicated by index arrays `INDX1`, ..., `INDXn`. Each element of the result is equal to the sum of the corresponding element of `BASE` and the elements of `ARRAY` scattered to that position.

Class. Transformational function.

Arguments.

`ARRAY`	must be of type integer, real, or complex. It must not be scalar.
`BASE`	must be of the same type and kind type parameter as `ARRAY`. It must not be scalar.
`INDX1,...,INDXn`	must be of type integer and conformable with `ARRAY`. The number of `INDX` arguments must be equal to the rank of `BASE`.
`MASK` (optional)	must be of type logical and must be conformable with `ARRAY`.

Result Type, Type Parameter, and Shape. Same as `BASE`.

Result Value. The element of the result corresponding to the element b of `BASE` has the value `SUM( (/`$a_1, a_2, ..., a_m, b$`/) )`, where $(a_1, \ldots, a_m)$ are the elements of `ARRAY` associated with b as described in Section 5.4.4.

Example. `SUM_SCATTER((/1, 2, 3, 1/), (/4, -5, 7/), (/1, 1, 2, 2/))` is $\begin{bmatrix} 7 & -1 & 7 \end{bmatrix}$.

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

Optional Arguments. DIM, MASK, SEGMENT, EXCLUSIVE

Description. Computes a reverse, segmented SUM scan along dimension DIM of ARRAY.

Class. Transformational function.

Arguments.

ARRAY	must be of type integer, real, or complex. It must not be scalar.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.
MASK (optional)	must be of type logical and must be conformable with ARRAY.
SEGMENT (optional)	must be of type logical and must have the same shape as ARRAY.
EXCLUSIVE (optional)	must be of type logical and must be scalar.

Result Type, Type Parameter, and Shape. Same as ARRAY.

Result Value. Element r of the result has the value SUM((/ $a_1, \ldots, a_m$ /)) where $(a_1, \ldots, a_m)$ is the (possibly empty) set of elements of ARRAY selected to contribute to r by the rules stated in Section 5.4.5.

Example. SUM_SUFFIX((/1,2,3,4,5/), SEGMENT= (/F,F,F,T,T/)) is $\begin{bmatrix} 6 & 5 & 3 & 9 & 5 \end{bmatrix}$.

Section 6

Extrinsic Procedures

This chapter defines the mechanism by which HPF programs may call non-HPF subprograms as *extrinsic procedures*. It provides the information needed to write an explicit interface for a non-HPF procedure. It defines the means for handling distributed and replicated data at the interface. This allows the programmer to use non-Fortran language facilities, perhaps to descend to a lower level of abstraction to handle problems that are not efficiently addressed by HPF, to hand-tune critical kernels, or to call optimized libraries. This interface can also be used to interface HPF to other languages, such as C.

> *Advice to implementors.* Annex A describes a suggested approach to supporting the coding of single-processor "node" code in single-processor Fortran 90 or in a single-processor subset of HPF; the idea is that only data that is mapped to a given physical processor is accessible to it. This allows the programming of MIMD multiprocessor machines in a single-program multiple-data (SPMD) style. (*End of advice to implementors.*)

6.1 Overview

It may be desirable for an HPF program to call a procedure written in a language other than HPF. Such a procedure might be written in any of a number of languages:

- A single-thread-of-control language not unlike HPF, where *one* copy of the procedure is conceptually executing and there is a single locus of control within the program text.
- A multiple-thread-of-control language, perhaps with dynamic assignment of loop iterations to processors or explicit dynamic process forking, where again there is, at least initially (upon invocation) *one* copy of the procedure that is conceptually executing but which may spawn multiple loci of control, possibly changing in number over time, within the program text.
- Any programming language targeted to a single processor, with the understanding that many copies of the procedure will be executed, one on each processor; this is frequently referred to as SPMD (Single Program, Multiple Data) style. We refer to a procedure written in this fashion as a *local* procedure.

A local procedure might be written in Fortran 77, Fortran 90, C, Ada, or Pascal, for example. A particularly interesting possibility is that a local procedure might be written in HPF! Not all HPF facilities may be used in writing local code, because some facilities address the question of executing on multiple processors and local code by definition runs on a single processor. See Annex A.

A called procedure that is written in a language other than HPF, whether or not it uses the local procedure execution model should be declared `EXTRINSIC` within an HPF program that calls it. The `EXTRINSIC` prefix declares what sort of interface should be used when calling indicated subprograms.

6.2 Definition and Invocation of Extrinsic Procedures

An explicit interface must be provided for each extrinsic procedure entry in the scope where it is called. This interface defines the "HPF view" of the extrinsic procedure.

H601 *extrinsic-prefix* **is** `EXTRINSIC (` *extrinsic-kind-keyword* `)`

H602 *extrinsic-kind-keyword* **is** `HPF`
or `HPF_LOCAL`
or `HPF_SERIAL`

An *extrinsic-prefix* may appear in a *subroutine-stmt* or *function-stmt* (as defined in the Fortran 90 standard) in the same place that the keyword `RECURSIVE` might appear. See Section 4.3 for the extended forms of the grammar rules for *function-stmt* and *subroutine-stmt* covering this case.

The *extrinsic-kind-keyword* indicates the kind of extrinsic interface to be used. (It may be helpful to think of this name as being to the subprogram calling interface what a `KIND` parameter is for a numeric type. However, an *extrinsic-kind* is not integer-valued; it is merely a keyword.) HPF defines three such keywords: `HPF`, `HPF_LOCAL`, and `HPF_SERIAL`. The keyword `HPF_LOCAL` is intended for use in calling routines coded in the "local HPF" style described in Annex A. The keyword `HPF` refers to the interface normally used for calling ordinary HPF routines.

Thus writing `EXTRINSIC(HPF)` in an HPF program has exactly the same effect as not using an `EXTRINSIC` specifier at all.

Thus writing `EXTRINSIC(HPF)` at the beginning of a *external-subprogram* in an HPF program has exactly the same effect as not using an `EXTRINSIC` specifier at all.

> *Rationale.* HPF defines the *extrinsic-kind-keyword* `HPF` primarily to set an example for other programming languages that might adopt this style of interface specification. For example, in an extended Fortran 90 compiler it would not be redundant to specify `EXTRINSIC(HPF)`, though it might be redundant to specify `EXTRINSIC(F90)`. In a C compiler it would not be redundant to specify `extrinsic(hpf)`. (*End of rationale.*)

A subprogram with an extrinsic interface lies outside the scope of HPF. However, explicit interfaces to such subprograms must conform to HPF. Note that any particular HPF implementation is free to support any selection of extrinsic kind keywords, or none at all except for `HPF` itself. Examples:

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

      INTERFACE OPERATOR (+)
        EXTRINSIC(C_LOCAL) FUNCTION LATKES(X, Y)   RESULT(Z)
          REAL, DIMENSION(:,:) :: X
          REAL, DIMENSION(SIZE(X,1), SIZE(X,2)) :: Y, Z
!HPF$     ALIGN WITH X :: Y, Z
!HPF$     DISTRIBUTE (BLOCK, BLOCK) X
        END FUNCTION
      END INTERFACE

      INTERFACE KNISH

        FUNCTION RKNISH(X)                          !normal HPF interface
           REAL X(:), RKNISH
        END RKNISH

        EXTRINSIC(SISAL) FUNCTION CKNISH(X)         !extrinsic interface
           COMPLEX X(:), CKNISH
        END CKNISH

      END INTERFACE
```

In the last interface block, two external procedures, one of them extrinsic and one not, are associated with the same generic procedure name, which returns a scalar of the same type as its array argument.

The intent is that a call to an extrinsic subprogram behaves, as observed by a calling program coded in HPF, exactly as if the subprogram has been coded in HPF.

> *Advice to implementors.* This is an obligation placed on the implementation of the interface and perhaps on the programmer when coding an extrinsic routine. However, it is also desirable to grant a certain freedom of implementation strategy so long as the obligation is satisfied. To this end an implementation may place certain restrictions on the programmer; moreover, each *extrinsic-kind-keyword* may call for a different set of restrictions.
>
> For example, an implementation on a parallel processor may find it convenient to replicate scalar arguments so as to provide a copy on every processor. This is permitted so long as this process is invisible to the caller. One way to achieve this is to place a restriction on the programmer: on return from the subprogram, all the copies of this scalar argument must have the same value. This implies that if the dummy argument has `INTENT(OUT)`, then all copies must have been updated consistently by the time of subprogram return. (*End of advice to implementors.*)

More generally, within a program unit of any given HPF kind, in order to call a subprogram of some other extrinsic kind, that subprogram must have an explicit interface; and the subprogram is expected to behave, as observed by the caller, roughly as if it had been written as code of the same extrinsic kind as the caller. Some of the responsibility for meeting this requirement may rest on the compiler and some on the programmer. Annex A, for example, spells out the responsbilities of the compiler and the programmer for calls from HPF code to HPF_LOCAL subprograms.

A particular restriction is placed on subprograms of extrinsic kind HPF_LOCAL: array dummy arguments of such subprograms must be declared as assumed-shape, both in the definition of the subprogram itself and in any interface blocks in other program units.

An *extrinsic-prefix* may also appear at the beginning of a *program-stmt*, *module-stmt*, or *block-data-stmt*.

H603	*program-stmt*	**is**	[*extrinsic-prefix*] PROGRAM *program-name*
H604	*module-stmt*	**is**	[*extrinsic-prefix*] MODULE *module-name*
H605	*block-data-stmt*	**is**	[*extrinsic-prefix*] BLOCK DATA *block-data-name*

Fortran 90 syntax rule R1102 (for *program-stmt*) is here rewritten as rule H603, rule R1105 (for *module-stmt*) is here rewritten as rule H604, and rule R1111 (for *block-data-stmt*) as rule H605.

Writing EXTRINSIC(HPF) at the beginning of *any* program unit of an HPF program has exactly the same effect as not using an EXTRINSIC specifier at all. Conversely, any program unit of an HPF program that has no *extrinsic-prefix* in its first statement is assumed to be of extrinsic kind HPF.

All extrinsic kind keywords whose names begin with the three letters "HPF" are reserved for present or future definition by this specification and its successors. A program unit whose extrinsic kind keyword begins with "HPF" is said to be "of an HPF extrinsic kind."

A main-program whose extrinsic kind is HPF_LOCAL or HPF_SERIAL behaves as if it were a subroutine of extrinsic kind HPF_LOCAL that is called with no arguments from a main program of extrinsic kind HPF whose executable part consists solely of that call.

Within any module of an HPF extrinsic kind, every *module-subprogram* must be of that same extrinsic kind and any module-subprogram whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind. Similarly, within any *main-program* or *external-subprogram* of an HPF extrinsic kind, every *internal-subprogram* must be of that same extrinsic kind and any *internal-subprogram* whose extrinsic kind is not given explicitly is assumed to be of that extrinsic kind.

A *function-stmt* or *subroutine-stmt* that appears within an *interface-block* within a program unit of an HPF extrinsic kind may have an extrinsic prefix mentioning any extrinsic kind supported by the language implementation; but if no *extrinsic-prefix* appears in such a *function-stmt* or *subroutine-stmt*, then it is assumed to be of the same HPF extrinsic kind as the host scoping unit.

The following sample code illustrates these rules:

```
      PROGRAM DUMPLING
  INTERFACE
    EXTRINSIC(HPF_LOCAL) SUBROUTINE GNOCCHI(P, L, X)
      INTERFACE
              SUBROUTINE P(Q)
```

		extrinsic kind of the used module		
		HPF	HPF_SERIAL	HPF_LOCAL
extrinsic kind	HPF	**T P D**	**T P**	**T P**
of the using	HPF_SERIAL	**T**	**T P D**	**T**
program unit	HPF_LOCAL	**T**	**T**	**T P D**

T = derived type definitions
P = procedures and procedure interfaces
D = data objects

Table 6.1: Entities that a using program unit is entitled to access from a module, according to the HPF extrinsic kind of each

```
                  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
```

The main program, DUMPLING, when compiled by an HPF compiler, is implicitly of extrinsic kind HPF. Interfaces are declared to three external subroutines GNOCCHI, POTSTICKER, and KNOEDEL. The first two are of extrinsic kind HPF_LOCAL and the third is of kind COBOL_LOCAL. Now GNOCCHI accepts two dummy procedure arguments and so interfaces must be declared for those. Because no *extrinsic-prefix* is given for dummy argument P, its extrinsic kind is that of its host scoping unit, the declaration of subroutine GNOCCHI, which ahs extrinsic kind HPF_LOCAL. The declaration of the corresponding actual argument POTSTICKER needs to have an explicit *extrinsic-prefix* because its host scoping unit is program DUMPLING, of extrinsic kind HPF.

If a module X of one HPF extrinsic kind is used from a program unit Y of another HPF extrinsic kind, then only names of items in X that Y is entitled to use or invoke may be imported; that is, either X makes private all items that Y is not entitled to use, or the USE statement in Y has an ONLY options that lists only names of items it is entitled to use.

A named COMMON block in any program unit of an HPF kind will be associated with the COMMON block, if any, of that same name in every other program unit of that same extrinsic kind; similarly for unnamed COMMON. (Such *COMMON* storage behaves as other declared data objects within program units of that extrinsic kind; in particular, for `HPF_LOCAL` code there will be one copy of the `COMMON` block on each processor.)

It is not permitted for any given `COMMON` block name to be used in program units of different HPF kinds within a single program; similarly, it is not permitted for unnamed `COMMON` to be used in program units of different HPF kinds within a single program.

Advice to implementors. (Implementors are advised to follow a similar rule for all extrinsic kind keywords, not just those starting with `HPF`.) (*End of advice to implementors.*)

6.3 Requirements on the Called Extrinsic Procedure

HPF requires a called extrinsic procedure to satisfy the following behavioral requirements:

1. The overall implementation must behave as if all actions of the caller preceding the subprogram invocation are completed before any action of the subprogram is executed; and as if all actions of the subprogram are completed before any action of the caller following the subprogram invocation is executed.

2. `IN`/`OUT` intent restrictions declared in the interface for the extrinsic subroutine must be obeyed.

3. Replicated variables, if updated, must be updated consistently. More precisely, if a variable accessible to a local subprogram has a replicated representation and is updated by (one or more copies of) the local subroutine, then all copies of the replicated data must have identical values when the last processor returns from the local procedure.

4. No HPF variable is modified unless it could be modified by an HPF procedure with the same explicit interface.

   Note in particular that even though an `HPF_LOCAL` routine is not permitted to access and modify HPF global data, other kinds of extrinsic routines may do so to the extent that an HPF procedure could.

5. When a subprogram returns and the caller resumes execution, all objects accessible to the caller after the call are mapped exactly as they were before the call.

   Advice to implementors.
   Note that, as with a non-extrinsic (that is, ordinary HPF) subprogram, actual arguments may be copied or remapped in any way, so long as the effect is undone on return from the subprogram.
   (*End of advice to implementors.*)

6. Exactly the same set of processors are visible to the HPF environment before and after the subprogram call.

The call to an extrinsic procedure that fulfills these rules is semantically equivalent to the execution of an ordinary HPF procedure.

Annex A has examples of the use of local subprograms through extrinsic interfaces.

Section 7

Storage and Sequence Association

HPF allows the mapping of variables across multiple processors in order to improve parallel performance. FORTRAN 77 and Fortran 90 both specify relationships between the storage for data objects associated through `COMMON` and `EQUIVALENCE` statements, and the order of array elements during association at procedure boundaries between actual arguments and dummy arguments. Otherwise, the location of data is not constrained by the language.

`COMMON` and `EQUIVALENCE` statements constrain the alignment of different data items based on the underlying model of storage units and storage sequences:

> *Storage association is the association of two or more data objects that occurs when two or more storage sequences share or are aligned with one or more storage units.*
>
> — Fortran Standard (14.6.3.1)

The model of storage association is a single linearly addressed memory, based on the traditional single address space, single memory unit architecture. This model can cause severe inefficiencies on architectures where storage for variables is mapped.

Sequence association refers to the order of array elements that Fortran requires when an array expression or array element is associated with a dummy array argument:

> *The rank and shape of the actual argument need not agree with the rank and shape of the dummy argument, ...*
>
> — Fortran Standard (12.4.1.4)

As with storage association, sequence association is a natural concept only in systems with a linearly addressed memory.

As an aid to porting FORTRAN 77 codes, HPF allows codes that rely on sequence and storage association to be valid in HPF. Some modification to existing FORTRAN 77 codes may nevertheless be necessary. This chapter explains the relationship between HPF data mapping and sequence and storage association.

7.1 Storage Association

7.1.1 Definitions

1. `COMMON` blocks are either *sequential* or *nonsequential*, as determined by either explicit directive or compiler default. A sequential `COMMON` block has a single common block storage sequence (5.5.2.1).

2. An *aggregate variable group* is a collection of variables whose individual storage sequences are parts of a single storage sequence.

   Variables associated by EQUIVALENCE statements or by a combination of EQUIVALENCE and COMMON statements form an aggregate variable group. The variables of a sequential COMMON block form a single aggregate variable group.

3. The *size* of an aggregate variable group is the number of storage units in the group's storage sequence (14.6.3.1).

4. If there is a member in an aggregate variable group whose storage sequence is totally associated (14.6.3.3) with the storage sequence of the aggregate variable group, that variable is called an *aggregate cover*.

5. Variables are either *sequential* or *nonsequential*. A variable is *sequential* if and only if any of the following holds:

   (a) it appears in a sequential COMMON block;
   (b) it is a member of an aggregate variable group;
   (c) it is an assumed-size array;
   (d) it is a component of a derived type with the Fortran 90 SEQUENCE attribute; or
   (e) it is declared to be sequential in an HPF SEQUENCE directive.

   A sequential variable can be storage associated or sequence associated; nonsequential variables cannot.

6. A COMMON block contains a sequence of *components*. Each component is either an aggregate variable group, or a variable that is not a member of any aggregate variable group. Sequential COMMON blocks contain a single component. Nonsequential COMMON blocks may contain several components that may be nonsequential or sequential variables or aggregate variable groups.

7. A variable is *explicitly mapped* if it appears in an HPF mapping directive within the scoping unit in which it is declared; otherwise it is *implicitly mapped*. A mapping directive is an ALIGN, or DISTRIBUTE, or REALIGN, or REDISTRIBUTE, or INHERIT, or DYNAMIC directive, or any directive that confers an alignment , a distribution, or the INHERIT or DYNAMIC attribute.

7.1.2 Examples of Definitions

```
      IMPLICIT REAL (A-Z)
      COMMON /FOO/ A(100), B(100), C(100), D(100), E(100)
      DIMENSION X(100), Y(150), Z(200)


!Example 1:
      EQUIVALENCE ( A(1), Z(1) )
!Four components: (A, B), C, D, E
!Sizes are: 200, 100, 100, 100
```

```
!Example 2:
      EQUIVALENCE ( B(100), Y(1) )
!Three components A, (B, C, D), E
!Sizes are: 100, 300, 100

!Example 3:
      EQUIVALENCE ( E(1), Y(1) )
!Five components: A, B, C, D, E
!Sizes are: 100, 100, 100, 100, 150

!Example 4:
      EQUIVALENCE ( A(51), X(1) ) ( B(100), Y(1) )
!Two components (A, B, C, D), E
!Sizes are: 400, 100

!Example 5:
      EQUIVALENCE ( A(51), X(1) ) ( C(80), Y(1) )
!Two components: (A, B), (C, D, E)
!Sizes are: 200, 300

!Example 6:
      EQUIVALENCE (Y(100), Z(1))
!One aggregate variable group (Y, Z), not involving the COMMON block.
!Size is 299

!Example 7:
!HPF$ SEQUENCE /FOO/
!The COMMON has one component, (A, B, C, D, E)
!Size is 500
```

In Examples 1–6, `COMMON` block `/FOO/` is nonsequential. Aggregate variable groups are shown as components in parentheses. Aggregate covers are `Z` in Example 1 and `Y` in Example 3.

7.1.3 Sequence Directives

A `SEQUENCE` directive is defined to allow a user to declare explicitly that variables or `COMMON` blocks are to be treated by the compiler as sequential. (`COMMON` blocks are by default nonsequential. Variables are nonsequential unless Definition 5 applies.) Some implementations may supply an optional compilation environment where the `SEQUENCE` directive is applied by default. For completeness in such an environment, HPF defines a `NO SEQUENCE` directive to allow a user to establish that the usual nonsequential default should apply to a scoping unit, or selected variables and `COMMON` blocks within the scoping unit.

H701	*sequence-directive*	**is**	`SEQUENCE` [[::] *association-name-list*]
		or	`NO SEQUENCE` [[::] *association-name-list*]
H702	*association-name*	**is**	*object-name*
		or	*function-name*
		or	`/` [*common-block-name*] `/`

Constraint: A variable or `COMMON` block name may appear at most once in a *sequence-directive* within any scoping unit.

Constraint: Only one sequence directive with a given *association-name* is permitted in the same scoping unit.

7.1.4 Storage Association Rules

1. A *sequence-directive* with an empty *association-name-list* is treated as if it contained the name of all implicitly mapped variables and `COMMON` blocks in the scoping unit which cannot otherwise be determined to be sequential or nonsequential by their language context.

2. A sequential variable may not be explicitely mapped unless it is a scalar or rank-one array, and is an aggregate cover. If there is more than one aggregate cover for an aggregate variable group, only one may be explicitly mapped.

3. No explicit mapping may be given for a component of a derived type having the Fortran 90 `SEQUENCE` attribute. In HPF 1, no components may have explicit mapping, but the consequence of Fortran 90 semantics are that even if, in some future version of HPF, components could have explicit mappings, those with the Fortran 90 `SEQUENCE` attribute may not.

4. No explicit mapping may be given for a dummy argument that is an assumed size array.

5. If a `COMMON` block is nonsequential, then all of the following must hold:

   (a) Every occurrence of the `COMMON` block has exactly the same number of components with each corresponding component having a storage sequence of exactly the same size;

   (b) If a component is a nonsequential variable in *any* occurrence of the `COMMON` block, then it must be nonsequential with identical type, shape, and mapping attributes in *every* occurrence of the `COMMON` block;

   (c) If a component is sequential and explicitly mapped (either a variable or an aggregate variable group with an explicitly mapped aggregate cover) in any occurrence of the `COMMON` block, then it must be sequential and explicitly mapped with identical mapping attributes in *every* occurrence of the `COMMON` block. In addition, the type and shape of the explicitly mapped variable must be identical in all occurrences; and

   (d) Every occurrence of the `COMMON` block must be nonsequential.

7.1.5 Storage Association Discussion

Advice to users. Under these rules, variables in a `COMMON` block can be mapped as long as the components of the `COMMON` block are the same in every scoping unit that declares the `COMMON` block. Rule 2 also allows variables involved in an `EQUIVALENCE` statement to be mapped by the mechanism of declaring a rank-one array to cover exactly the aggregate variable group and mapping that array.

Since an HPF program is nonconforming if it specifies any mapping that would cause a scalar data object to be mapped onto more than one abstract processor, there is a constraint on the sequential variables and aggregate covers that can be mapped. In particular, programs that direct double precision or complex arrays to be mapped such that the storage units of a single array element are split because of some `EQUIVALENCE` statement or `COMMON` block layout are nonconforming.

Correct FORTRAN 77 or Fortran 90 programs will not necessarily be correct without modification in HPF. As the examples in the next section illustrate, use of `EQUIVALENCE` with `COMMON` blocks can impact mappability of the variables in subtle ways. To allow maximum optimization for performance, the HPF default for variables is to consider them mappable. In order to get correct separate compilation for subprograms that use `COMMON` blocks with different aggregate variable groups in different scoping units, it will be necessary to insert the HPF `SEQUENCE` directive.

As a check-list for a user to determine the status of a variable or `COMMON` block, the following questions can be applied, in order:

- Does the variable appear in some explicit language context which dictates sequential (e.g. `EQUIVALENCE`) or nonsequential (e.g. array-valued function result variable)?
- If not, does the variable appear in an explicit mapping directive?
- If not, does the variable or `COMMON` block name appear in the list of names on a `SEQUENCE` or `NO SEQUENCE` directive?
- If not, does the scoping unit contain a nameless `SEQUENCE` or `NO SEQUENCE`?
- If not, is the compilation affected by some special implementation-dependent environment which dictates that names default to `SEQUENCE`?
- If not, then the compiler will consider the variable or `COMMON` block name nonsequential and is free to apply data mapping optimizations disregarding Fortran sequence and storage association.

(*End of advice to users.*)

Advice to implementors. In order to protect the user and to facilitate portability of older codes, two implementation options are strongly recommended. First, every implementation should supply some mechanism to verify that the type and shape of every mappable array and the sizes of aggregate variable groups in `COMMON` blocks are the same in every scoping unit unless the `COMMON` blocks are declared to be sequential. This same check should also verify that identical mappings have been selected for the variables in `COMMON` blocks. Implementations without interprocedural information can use a link-time check. The second implementation option recommended is a mechanism to declare that variables and `COMMON` blocks for a given compilation should be considered sequential unless declared otherwise. The purpose of this feature is to permit compilation of large old libraries or subprograms where storage association is known to exist without requiring that the code be modified to apply the HPF `SEQUENCE` directive to every `COMMON` block. (*End of advice to implementors.*)

7.1.6 Examples of Storage Association

```
      IMPLICIT REAL (A-Z)
      COMMON /FOO/ A(100), B(100), C(100), D(100), E(100)
      DIMENSION X(100), Y(150), Z(200), ZZ(300)

      EQUIVALENCE ( A(1), Y(1) )
!Aggregate variable group is not mappable.
!Sizes are: 200, 100, 100, 100.

      EQUIVALENCE ( B(100), Y(1) ), ( B(1), ZZ(1) )
!Aggregate variable group is mappable only by mapping ZZ.
!ZZ is an aggregate cover for B, C, D, and Y.
!Sizes are: 100, 300, 100.

      EQUIVALENCE ( E(1), Y(1) )
!Aggregate variable group is mappable by mapping Y.
!Sizes are: 100, 100, 100, 100, 150.

      COMMON /TWO/ A(20,40),E(10,10),G(10,100,1000),H(100),P(100)
      REAL COVER(200)
      EQUIVALENCE (COVER(1), H(1))
!HPF$   SEQUENCE A
!HPF$   ALIGN E ...
!HPF$   DISTRIBUTE COVER (CYCLIC(2))
```

Here `A` is sequential and implicitly mapped, `E` is explicitly mapped, `G` is implicitly mapped, the aggregate cover of the aggregate variable group `(H, P)` is explicitly mapped. `/TWO/` is a nonsequential `COMMON` block.

In another subprogram, the following declarations may occur:

```
      COMMON /TWO/ A(800), E(10,10), G(10,100,1000), Z(200)
!HPF$   SEQUENCE A, Z
!HPF$   ALIGN E ...
!HPF$   DISTRIBUTE Z (CYCLIC(2))
```

There are four components of the same size in both occurrences. Components one and four are sequential. Components two and four are explicitly mapped, with the same type, shape and mapping attributes.

The first component, `A`, must be declared sequential in both occurrences because its shape is different. It may not be explicitly mapped in either because it is not rank-one or scalar in the first.

`E` and `G` must agree in type and shape in both occurrences. `E` must have the same explicit mapping and `G` must have no explicit mapping in both occurrences, since they are nonsequential variables.

The fourth component must have the same explicit mapping in both occurrences, and must be made sequential explicitly in the second.

7.2 Argument Passing and Sequence Association

For actual arguments in a procedure call, Fortran 90 allows an array element (scalar) to be associated with a dummy argument that is an array. It furthermore allows the shape of a dummy argument to differ from the shape of the corresponding actual array argument, in effect reshaping the actual argument via the subroutine call. Storage sequence properties of Fortran are used to identify the values of the dummy argument. This feature, carried over from FORTRAN 77, has been widely used to pass starting addresses of subarrays, rows or columns of a larger array, to procedures. For HPF arrays that are potentially mapped across processors, this feature is not fully supported.

7.2.1 Sequence Association Rules

1. When an array element or the name of an assumed-size array is used as an actual argument, the associated dummy argument must be a scalar or specified to be a sequential array.

   An array-element designator of a nonsequential array must not be associated with a dummy array argument.

2. When an actual argument is an array or array section and the corresponding dummy argument differs from the actual argument in shape, then the dummy argument must be declared sequential and the actual array argument must be sequential.

3. A variable of type character (scalar or array) is nonsequential if it conforms to the requirements of Definition 5 of Section 7.1.1. If the length of an explicit-length character dummy argument differs from the length of the actual argument, then both the actual and dummy arguments must be sequential.

4. Without an explicit interface, a sequential actual may not be associated with a nonsequential dummy and a nonsequential actual may not be associated with a sequential dummy.

7.2.2 Discussion of Sequence Association

When the shape of the dummy array argument and its associated actual array argument differ, the actual argument must not be an expression. There is no HPF mechanism for declaring that the value of an array-valued expression is sequential. In order to associate such an expression as an actual argument with a dummy argument of different rank, the actual argument must first be assigned to a named array variable that is forced to be sequential according to Definition 5 of Section 7.1.1.

7.2.3 Examples of Sequence Association

Given the following subroutine fragment:

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

By rule 1

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

is legal only if X is declared sequential in HOME and ET is sequential in the calling routine.

Likewise, by rule 2 and 4

```
CALL HOME (ET)
```

requires either that ET and X are both sequential arrays or that ET and X have the same shape and have the same sequence attribute.

Rule 3 addresses a special consideration for variables of type character. Change of the length of character variables across a call, as in

```
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)
    !Note that short_dictionary(3) is 'agog', for example
```

is conceptually legal in FORTRAN 77 and Fortran 90. In HPF, both the actual argument and dummy argument must be sequential. (By the way, "Chargoggagoggmanchaugagoggchaubunagungamaugg" is the original Nipmuc name for what is now called "Lake Webster" in Massachusetts.)

Section 8

Subset High Performance Fortran

This chapter presents a subset of HPF capable of being implemented more rapidly than the full HPF. A subset implementation will provide a portable interim HPF capability. Full HPF implementations should be developed as rapidly as possible. The definition of the subset language is intended to be a minimal requirement. A given implementation may support additional Fortran 90 and HPF features.

8.1 Fortran 90 Features in Subset High Performance Fortran

The items listed here are the features of the HPF subset language. 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. (See Section 7 for detailed discussion of the exception.)
- 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 now have been extended 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)

8.2 Discussion of the Fortran 90 Subset Features

Rationale. There are many Fortran 90 features which are useful and relatively easy to implement, but are not included in the subset language. Features were selected for the subset language for several reasons.

The MIL-STD-1753 features have been implemented so widely that many users have forgotten that they are not part of FORTRAN 77. They are included in the HPF subset.

The biggest addition to FORTRAN 77 in the HPF subset language is the inclusion of the array language. A number of vendors have identified the usefulness of array operations for concise expression of parallelism and already support these features. However, the character array language is not part of the subset.

The new storage classes such as allocatable, automatic, and assumed-shape objects are included in the subset. They provide an important alternative to the use of storage association features such as `EQUIVALENCE` for memory management.

Interface blocks have been added to the subset in order to facilitate use of the HPF directives across subroutine boundaries. The interface blocks provide a mechanism to specify the expected mapping of data, in addition to the types and intents of the arguments.

There were other Fortran 90 features considered for the subset. Some features such as `CASE` or `NAMELIST` were recognized as popular features of Fortran 90, but had no direct bearing on high performance. Other features such as support for double precision complex (via `KIND`) or procedureless `MODULES` were rejected because of the perception that the additional implementation complexity might delay release of subset compilers. It was not a goal of HPFF to define an "ideal" subset of Fortran 90 for all purposes.

Additional syntactic improvements are included, such as long names and the "!" form of comments, because of their general usefulness in program documentation, including the description of HPF itself. (*End of rationale.*)

8.3 HPF Features Not in Subset High Performance Fortran

All HPF directives and language extensions are included in the HPF subset language with the following exceptions:

- The `REALIGN`, `REDISTRIBUTE`, and `DYNAMIC` directives;
- The `INHERIT` directive.
- The `PURE` function attribute;
- The *forall-construct*;
- The HPF library and the `HPF_LIBRARY` module;
- Actual argument expressions corresponding to optional `DIM` arguments to the Fortran 90 `MAXLOC` and `MINLOC` intrinsic functions that are not initialization expressions; and
- The `EXTRINSIC` function attribute.

8.4 Discussion of the HPF Extension Subset

Rationale. The data mapping features of the HPF subset are limited to static mappings, plus the possible remapping of arguments across the interface of subprogram boundaries. Since the subset language does not include `MODULES`, and `COMMON` block variables cannot be remapped, this restriction only impacts remapping of local variables and additional remapping of arguments, after the subprogram boundary.

The `INHERIT` directive is no longer included in the subset. The case where it is most useful (to describe the template of the full array, when only a section of an array is passed as an argument) cannot not be declared properly with the former restriction on use of transcriptive distributions, combined with the fact that processor directives cannot be used to describe only parts of the processor set.

Only the simplest version of `FORALL` statement is required in the subset. Note that the omission of the `PURE` attribute from the subset means that only HPF and Fortran 90 intrinsic functions can be called from the `FORALL` statement. No other subprograms can be called.

Only the intrinsics which are useful for declaration of variables and mapping inquiries are included in the subset. The full set of extended operations proposed for the HPF library is not required and since `MODULE` is not part of the subset, the `HPF_LIBRARY` module is also not part of the subset. The extrinsic interface attribute is also not in the subset. This includes any specific extrinsic models such as the model described in the Annex A.

All of these HPF language reductions are made in the spirit of allowing vendors to produce a usable subset version of HPF quickly so that initial experimentation with the language can begin. This list of HPF features excluded from the subset should not be interpreted as requiring implementors to omit the features from the subset. Implementations with as many HPF features as possible are encouraged. The list does, however, establish the features a user should avoid if an HPF application is expected to be moved between different HPF subset implementations. (*End of rationale.*)

Annex A

Coding Local Routines in HPF and Fortran 90

This annex defines a mechanism for coding single-processor "node" code in single-processor Fortran 90 or in a single-processor subset of HPF; the idea is that only data that is mapped to a given physical processor is accessible to it. This allows the programming of MIMD multiprocessor machines in a single-program multiple-data (SPMD) style. Implementation-specific libraries may be provided to facilitate communication between the physical processors that are independently executing this code, but the specification of such libraries is outside the scope of HPF and outside the scope of this annex.

The `EXTRINSIC` mechanism, which allows an HPF programmer to declare a calling interface to a non-HPF subprogram, is described in Section 6 of the HPF specification.

From the caller's standpoint, an invocation of an extrinsic procedure from a "global" HPF program has the same semantics as an invocation of a regular procedure. The callee may see a different picture. This annex describes a particular set of conventions for coding callees in the "local" style in which a copy of the subprogram executes on each processor (of which there may be one or many).

An extrinsic procedure can be defined as explicit SPMD code by specifying the local procedure code that is to execute on each processor. HPF provides a mechanism for defining local procedures in a subset of HPF that excludes only data mapping directives, which are not relevant to local code. If a subprogram definition or interface uses the *extrinsic-kind-keyword* `HPF_LOCAL`, then an HPF compiler should assume that the subprogram is coded as a local procedure. Because local procedures written in HPF are thus syntactically distinguished, they may be intermixed unambiguously with global HPF code if the implementor of an HPF language processor chooses to support such intermixing.

This annex is divided into three parts:

1. The contract between the caller and a callee that is a local procedure, that is, defined as explicit Single Program Multiple Data (SPMD) code.

2. A specific version of this interface for the case where the callee is a local procedure coded in HPF (*extrinsic-kind-keyword* `HPF_LOCAL`). Such local procedures may be compiled separately or included as part of the text of a global HPF program.

3. A specific version of this interface for the case where extrinsic procedures are defined as explicit SPMD code with each local procedure coded in Fortran 90 (the *extrinsic-kind-keyword* might be, for instance, `F90_LOCAL`). Ideally these local procedures may

be separately compiled by a Fortran 90 compiler and then linked with HPF code, though this depends on implementation details.

A.1 Conventions for Local Subprograms

All HPF arrays accessible to an extrinsic procedure (arrays passed as arguments) are logically carved up into pieces; the local 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.

It is important not to confuse the extrinsic procedure, which is conceptually a single procedural entity called from the HPF program, with the local procedures, which are executed on each node, one apiece. An *invocation* of an extrinsic procedure results in a separate invocation of a local procedure on each processor. The *execution* of an extrinsic procedure consists of the concurrent execution of a local procedure on each executing processor. Each local procedure may terminate at any time by executing a `RETURN` statement. However, the extrinsic procedure as a whole terminates only after every local procedure has terminated; in effect, the processors are synchronized before return to a global HPF caller.

It is technically feasible to define extrinsic procedures in any other parallel language that maps to this basic SPMD execution model, or in any sequential language, including single-processor Fortran 90, with the understanding that one copy of the sequential code is executed on each processor. The extrinsic procedure interface is designed to ease implementation of local procedures in languages other than HPF; however, it is beyond the scope of the HPF specification or this annex to dictate implementation requirements for such languages or implementations. Nevertheless, a suggested way to use Fortran 90 to define local procedures is discussed in Section A.3.

With the exception of returning from a local procedure to the global caller that initiated local execution, there is no implicit synchronization of the locally executing processors. A local procedure may use any control structure whatsoever. To access data outside the processor requires either preparatory communication to copy data into the processor before running the local code, or communication between the separately executing copies of the local procedure. Individual implementations may provide implementation-dependent means for communicating, for example through a message-passing library or a shared-memory mechanism. Such communication mechanisms are beyond the scope of this specification. Note, however, that many useful portable algorithms that require only independence of control structure can take advantage of local routines, without requiring a communication facility.

This model assumes only that array axes are mapped independently to axes of a rectangular processor grid, each array axis to at most one processor axis (no "skew" distributions) and no two array axes to the same processor axis. This restriction suffices to ensure that each physical processor contains a subset of array elements that can be locally arranged in a rectangular configuration. (Of course, to compute the global indices of an element given its local indices, or vice versa, may be quite a tangled computation—but it will be possible.)

It is recommended that if, in any given implementation, an interface kind does not obey the conventions described in the section, then the name of that interface kind should not end in "`_LOCAL`".

A.1.1 Conventions for Calling Local Subprograms

The default mapping of scalar dummy arguments and of scalar function results is such that the argument is replicated on each physical processor. These mappings may, optionally, be explicit in the interface, but any other explicit mapping is not HPF conforming.

As in the case of non-extrinsic subprograms, actual arguments may be mapped in any way; if necessary, they are copied automatically to correctly mapped temporaries before invocation of and after return from the extrinsic procedure.

A.1.2 Calling Sequence

The actions detailed below have to occur prior to the invocation of the local procedure on each processor. These actions are enforced by the compiler of the calling routine, and are not the responsibility of the programmer, nor do they impact the local procedure. (The next section discusses restrictions on the local procedure.)

1. The processors are synchronized. In other words, all actions that logically precede the call are completed.

2. Each actual argument is remapped, if necessary, according to the directives (explicit or implicit) in the declared interface for the extrinsic procedure. Thus, HPF mapping directives appearing in the interface are binding—the compiler must obey these directives in calling local extrinsic procedures. (The reason for this rule is that data mapping is explicitly visible in local routines). Actual arguments corresponding to scalar dummy arguments are replicated (by broadcasting, for example) in all processors.

3. If a variable accessible to the called routine has a replicated representation, then all copies are updated prior to the call to contain the correct current value according to the sequential semantics of the source program.

After these actions have occurred, the local procedure is invoked on each processor. The information available to the local invocation is described below in Section A.1.3.

The following actions must occur before control is transferred back to the caller.

1. All processors are synchronized after the call. In other words, execution of every copy of the local routine is completed before execution in the caller is resumed.

2. The original distribution of arguments (and of the result of an extrinsic function) is restored, if necessary.

Advice to implementors. An implementation might check, before returning from the local subprogram, to make sure that replicated variables have been updated consistently by the subprogram. However, there is certainly no requirement—perhaps not even any encouragement—to do so. This is merely a tradeoff between speed and, for instance, debuggability. (*End of advice to implementors.*)

A.1.3 Information Available to the Local Procedure

The local procedure invoked on each processor is passed a *local argument* for each *global argument* passed by the caller to the (global) extrinsic procedure interface. Each global argument is a distributed HPF array or a replicated scalar. The corresponding local argument is the part of the global array stored locally, or the local copy of a scalar argument. An array actual argument passed by an HPF caller is called a *global array*; the subgrid of that global array passed to one copy of a local routine (because it resides in that processor) is called a *local array*.

If the extrinsic procedure is a function, then the local procedure is also a function. Each local invocation of that function will return the local part of the extrinsic function return value. If the extrinsic function is scalar-valued then the implicit mapping of the return value is replicated. Thus, all local functions must return the same value. If one desires to return one, possibly distinct, value per processor, then the extrinsic function must be declared to return a distributed rank-one array of size `NUMBER_OF_PROCESSORS`.

The run-time interface should provide enough information that each local function can discover for each local argument the mapping of the corresponding global argument, translate global indices to local indices, and vice-versa. A specific set of procedures that provide this information is listed in Section A.2.3. The manner in which this information is made available to the local routine depends on the implementation and the programming language used for the local routine.

A.2 Local Routines Written in HPF

This section provides a specific design for providing the required information to local procedures in the case these procedures are written in HPF.

Local procedures may be declared within an HPF program (and be compiled by an HPF compiler). The *subroutine-stmt* or *function-stmt* that begins the subprogram must contain the prefix `EXTRINSIC(HPF_LOCAL)`.

A.2.1 Restrictions

There are some restrictions on what HPF features may be used in writing a local, per-processor procedure.

A local HPF program unit may invoke other local program units or internal procedures, but it may not invoke an ordinary, "global" HPF routine. If a global HPF program calls local subprogram `A` with an actual array argument `X`, and `A` receives a portion of array `X` as dummy argument `P`, then `A` may call another local subprogram `B` and pass `P` or a section of `P` as an actual argument to `B`.

A local HPF program unit may not access global HPF data other than data that is accessible, either directly or indirectly, via the actual arguments. In particular, a local HPF program unit does not have access to global HPF `COMMON` blocks; `COMMON` blocks appearing in local HPF program units are not identified with global HPF `COMMON` blocks. The same name may not be used to identify a `COMMON` block both within a local HPF program unit and an HPF program unit in the same executable program.

Local program units can use all HPF constructs except for `DISTRIBUTE`, `REDISTRIBUTE`, `ALIGN`, `REALIGN`, and `INHERIT` directives.

Local program units can use all HPF constructs except for REDISTRIBUTE and REALIGN. Moreover, DISTRIBUTE, ALIGN, and INHERIT directives may be applied only to dummy arguments and function results; that is, every alignee and distributee must be a dummy argument or function result and every align-target must be a template, dummy argument, or function result. Mapping directives in local HPF program units are understood to have global meaning, as if they had appeared in global HPF code, applying the the global array of which a portion is passed on each processor. (The principal use of such mapping directives is in an HPF_LOCAL module that is used by a global HPF module.)

The distribution query library subroutines HPF_ALIGNMENT, HPF_TEMPLATE, and HPF_DISTRIBUTION may be applied to local arrays. Their outcome is the same as for a global array that happens to have all its elements on a single node.

Scalar dummy arguments must be mapped so that each processor has a copy of the argument. This holds true, by convention, if no mapping is specified for the argument in the interface. Thus, the constraint disallows only explicit alignment and distribution directives in an explicit interface that imply that a scalar dummy argument is not replicated on all processors.

An EXTRINSIC(HPF_LOCAL) routine may not be RECURSIVE.

An EXTRINSIC(HPF_LOCAL) routine may not have alternate returns.

An EXTRINSIC(HPF_LOCAL) routine may not be invoked, either directly or indirectly, in the body of a FORALL construct or in the body of an INDEPENDENT loop.

The attributes (type, kind, rank, optional, intent) of the dummy arguments must match the attributes of the corresponding dummy arguments in the explicit interface. A dummy argument of an EXTRINSIC(HPF_LOCAL) routine may not be a procedure name.

A dummy argument of an EXTRINSIC(HPF_LOCAL) routine may not have the POINTER attribute.

A dummy argument of an EXTRINSIC(HPF_LOCAL) routine must be nonsequential.

A dummy array argument of an EXTRINSIC(HPF_LOCAL) routine must have assumed shape, even when it is explicit shape in the interface. Note that, in general, the shape of a dummy array argument differs from the shape of the corresponding actual argument, unless there is a single executing processor.

Explicit mapping directives for dummy arguments and function result variables may not appear in a local procedure, although they may appear (in the case of the result of an array-valued function, they must appear) in the required explicit interface accessible to the caller.

Explicit mapping directives for dummy arguments and function result variables may appear in a local procedure. Such directives are understood as applying to the global array whose local sections are passed as actual arguments or results on each processor. If such directives appear, corresponding mapping directives must be visible to every global HPF caller. This may be done by providing an interface block in the caller, or by placing the local procedure in a module of extrinsic kind HPF_LOCAL that is then used by the global HPF program unit that calls the local procedure.

A local procedure may have several ENTRY points. A global HPF caller must contain a separate extrinsic interface for each entry point that can be invoked from the HPF program.

The behavior of I/O statements in a local procedure is implementation-dependent.

A.2.2 Argument Association

If a dummy argument of an EXTRINSIC(HPF_LOCAL) routine is an array, then the corresponding dummy argument in the specification of the local procedure must be an array of the same rank, type, and type parameters. When the extrinsic procedure is invoked, the local dummy argument is associated with the local array that consists of the subgrid of the global array that is stored locally. This local array will be a valid HPF array.

If a dummy argument of an EXTRINSIC(HPF_LOCAL) routine is a scalar then the corresponding dummy argument of the local procedure must be a scalar of the same type. When the extrinsic procedure is invoked then the local procedure is passed an argument that consists of the local copy of the replicated scalar. This copy will be a valid HPF scalar.

If an EXTRINSIC(HPF_LOCAL) routine is a function, then the local procedure is a function that returns a scalar of the same type and type parameters, or an array of the same rank, type, and type parameters, as the HPF extrinsic function. The value returned by each local invocation is the local part of the value returned by the HPF invocation.

Each physical processor has at most one copy of each HPF variable.

Consider the following extrinsic interface:

```
      INTERFACE
        EXTRINSIC(HPF_LOCAL) FUNCTION MATZOH(X, Y) RESULT(Z)
          REAL, DIMENSION(:,:) :: X
          REAL, DIMENSION(SIZE(X,1)) :: Y, Z
!HPF$     ALIGN WITH X(:,*) :: Y(:), Z(:)
!HPF$     DISTRIBUTE X(BLOCK, CYCLIC)
        END FUNCTION
      END INTERFACE
```

The corresponding local HPF procedure is specified as follows.

```
      EXTRINSIC(HPF_LOCAL) FUNCTION MATZOH(XX, YY) RESULT(ZZ)
      REAL, DIMENSION(:,:) :: XX
      REAL, DIMENSION(5 : SIZE(XX,1)+4) :: YY, 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
```

Assume that the function is invoked with an actual (global) array X of shape 3×3 and an actual vector Y of length 3 on a 4-processor machine, using a 2×2 processor arrangement (assuming one abstract processor per physical processor).

Then each local invocation of the function MATZOH receives the following actual arguments:

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)`

Here are the values to which each processor would set `NX1`, `LX1`, `UX1`, `NX2`, `LX2`, `UX2`, `NY`, `LY`, and `UY`:

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

The return array `ZZ` is distributed identically to `YY`: Processors (1,1) and (1,2) should return identical rank one arrays of size 2; processors (2,1) and (2,2) should return identical rank one arrays of size 1.

An actual argument to an extrinsic procedure may be a pointer. Since the corresponding dummy argument may not have the `POINTER` attribute, the dummy argument becomes associated with the target of the HPF global pointer. In no way may a local pointer become pointer associated with a global HPF target. Therefore, an actual argument may not be of a derived-type containing a pointer component.

> *Rationale.* It is expected that global pointer variables will have a different representation from that of local pointer variables, at least on distributed memory machines, because of the need to carry additional information for global addressing. This restriction could be lifted in the future. (*End of rationale.*)

Other inquiry intrinsics, such as `ALLOCATED` or `PRESENT`, should also behave as expected. Note that when a global array is passed to a local routine, some processors may receive an empty subarray. Such argument is `PRESENT` and has `SIZE` zero.

A.2.3 HPF Local Routine Library

Local HPF procedures can use any HPF intrinsic or library procedure.

> *Advice to implementors.* The arguments to such procedures will be local arrays. Depending on the implementation, the actual code for the intrinsic and library routines used by local HPF procedures may or may not be the same code used when called from global HPF code.
>
> (*End of advice to implementors.*)

In addition, local library procedures GLOBAL_ALIGNMENT, GLOBAL_DISTRIBUTION, and GLOBAL_TEMPLATE are provided to query the global mapping of an actual argument to an extrinsic function. Other local library procedures are provided to query the size, shape, and array bounds of an actual argument. These library procedures take as input the name of a dummy argument and return information on the corresponding global HPF actual argument. They may be invoked only by a local procedure that was directly invoked by global HPF code. If module facilities are available, they reside in a module called HPF_LOCAL_LIBRARY; a local routine that calls them should include the statement

```
USE HPF_LOCAL_LIBRARY
```

or some functionally appropriate variant thereof.

The local HPF library also provides a new derived type PROCID, to be used for processor identifiers. Each physical processor has a distinct identifier of type PROCID. It is assumed that a function is available to find the identifier of each executing processor—the syntax for calling such a function is beyond the scope of this document.

Advice to implementors.

It is likely that in many implementations type PROCID will be effectively identical to type INTEGER.

(*End of advice to implementors.*)

The local HPF library identifies each physical processor by an integer in the range 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 calling processor.

A.2.3.1 Accessing Dummy Arguments by Blocks

The mapping of a global HPF array to the physical processors places one or more *blocks*, which are groups of elements with consecutive indices, on each processor. The number of blocks mapped to a processor is the product of the number of blocks of consecutive indices in each dimension that are mapped to it. For example, a rank-one array X with a CYCLIC(4) distribution will have blocks containing four elements, except for a possible last block having $1 + \texttt{SIZE(X)} \bmod 4$ elements. On the other hand, if X is first aligned to a template or an array having a CYCLIC(4) distribution, and a non-unit stride is employed (as is !HPF$ ALIGN X(I) WITH T(3*I)), then its blocks may have fewer than four elements. In this case, when the align stride is three and the template has a block-cyclic distribution with four template elements per block, the blocks of X have either one or two elements each. If the align stride were five, then all blocks of X would have exactly one element, as template blocks to which no array element is aligned are not counted in the reckoning of numbers of blocks.

The portion of a global array argument associated with a dummy argument in an HPF_LOCAL subprogram may be accessed in a block-by-block fashion. Three of the local library routines, LOCAL_BLKCNT, LOCAL_LINDEX, and LOCAL_UINDEX, allow easy access to the local storage of a particular block. Their use for this purpose is illustrated by the following example, in which the local data are initialized one block at a time:

```
      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
```

A.2.3.2 GLOBAL_ALIGNMENT(ARRAY, ...)

This has the same interface and behavior as the HPF inquiry subroutine `HPF_ALIGNMENT`, but it returns information about the *global* HPF array actual argument associated with the local dummy argument `ARRAY`, rather than returning information about the local array.

A.2.3.3 GLOBAL_DISTRIBUTION(ARRAY, ...)

This has the same interface and behavior as the HPF inquiry subroutine `HPF_DISTRIBUTION`, but it returns information about the *global* HPF array actual argument associated with the local dummy argument ARRAY, rather than returning information about the local array.

A.2.3.4 GLOBAL_TEMPLATE(ARRAY, ...)

This has the same interface and behavior as the HPF inquiry subroutine HPF_TEMPLATE, but it returns information about the *global* HPF array actual argument associated with the local dummy argument ARRAY, rather than returning information about the local array.

A.2.3.5 GLOBAL_LBOUND(ARRAY, DIM)

Optional argument. DIM

Description. Returns all the lower bounds or a specified lower bound of the actual HPF global array argument associated with an HPF_LOCAL dummy array argument.

Class. Inquiry function.

Arguments.

ARRAY	may be of any type. It must not be a scalar. It must be a dummy array argument of an HPF_LOCAL procedure which is argument associated with a global HPF array actual argument.
DIM (optional)	must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter and Shape. The result is of type default integer. It is scalar if DIM is present; otherwise the result is an array of rank one and size n, where n is the rank of ARRAY.

Result Value.

Case (i): If the actual argument associated with the actual argument associated with ARRAY is an array section or an array expression, other than a whole array or an array structure component, GLOBAL_LBOUND(ARRAY, DIM) has the value 1; otherwise it has a value equal to the lower bound for subscript DIM of the actual argument associated with the actual argument associated with ARRAY.

Case (ii): GLOBAL_LBOUND(ARRAY) has a value whose ith component is equal to GLOBAL_LBOUND(ARRAY, i), for $i = 1, 2, \ldots n$ where n is the rank of ARRAY.

Examples. Assuming A is declared by the statement

```
INTEGER A(3:100, 200)
```

and is argument associated with B, the value of GLOBAL_LBOUND(B) is $\begin{bmatrix} 3 & 1 \end{bmatrix}$. If B is argument associated with the section, A(5:10, 10), the value of GLOBAL_LBOUND(B, 1) is 1.

A.2.3.6 GLOBAL_SHAPE(SOURCE)

Description. Returns the shape of the global HPF actual argument associated with an array or scalar dummy argument of an HPF_LOCAL procedure.

Class. Inquiry function.

Argument.

`SOURCE` may be of any type. It may be array valued or a scalar. It must be a dummy argument of an HPF_LOCAL procedure which is argument associated with a global HPF actual argument.

Result Type, Type Parameter and Shape. The result is a default integer array of rank one whose size is equal to the rank of `SOURCE`.

Result Value. The value of the result is the shape of the global actual argument associated with the actual argument associated with `SOURCE`.

Examples. Assuming `A` is declared by the statement

```
INTEGER A(3:100, 200)
```

and is argument associated with `B`, the value of `GLOBAL_SHAPE(B)` is $\left[\begin{array}{cc} 98 & 200 \end{array}\right]$. If `B` is argument associated with the section, `A(5:10, 10)`, the value of `GLOBAL_SHAPE(B)` is $\left[\begin{array}{c} 6 \end{array}\right]$.

A.2.3.7 GLOBAL_SIZE(ARRAY, DIM)

Optional argument. `DIM`

Description. Returns the extent along a specified dimension of the global HPF actual array argument associated with a dummy array argument of an HPF_LOCAL procedure.

Class. Inquiry function.

Argument.

`ARRAY` may be of any type. It must not be a scalar. It must be a dummy argument of an HPF_LOCAL procedure which is argument associated with a global HPF actual argument.

`DIM` (optional) must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.

Result Type, Type Parameter and Shape. Default integer scalar.

Result Value. The result has a value equal to the extent of dimension `DIM` of the actual argument associated with the actual argument associated with `ARRAY` or, if `DIM` is absent, the total number of elements in the actual argument associated with the actual argument associated with `ARRAY`.

Examples. Assuming `A` is declared by the statement

```
INTEGER A(3:10, 10)
```

and is argument associated with `B`, the value of `GLOBAL_SIZE(B, 1)` is 8. If `B` is argument associated with the section, `A(5:10, 2:4)`, the value of `GLOBAL_SIZE(B)` is 18.

A.2.3.8 GLOBAL_UBOUND(ARRAY, DIM)

Optional argument. `DIM`

Description. Returns all the upper bounds or a specified upper bound of the actual HPF global array argument associated with an HPF_LOCAL dummy array argument.

Class. Inquiry function.

Arguments.

`ARRAY`	may be of any type. It must not be a scalar. It must be a dummy array argument of an HPF_LOCAL procedure which is argument associated with a global HPF array actual argument.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

Result Type, Type Parameter and Shape. The result is of type default integer. It is scalar if `DIM` is present; otherwise the result is an array of rank one and size n, where n is the rank of `ARRAY`.

Result Value.

Case (i): If the actual argument associated with the actual argument associated with `ARRAY` is an array section or an array expression, other than a whole array or an array structure component, `GLOBAL_UBOUND(ARRAY, DIM)` has a value equal to the number of elements in the given dimension; otherwise it has a value equal to the upper bound for subscript `DIM` of the actual argument associated with the actual argument associated with `ARRAY`, if dimension `DIM` does not have size zero and has the value zero if dimension `DIM` has size zero.

Case (ii): `GLOBAL_UBOUND(ARRAY)` has a value whose ith component is equal to `GLOBAL_UBOUND(ARRAY,` i`)`, for $i = 1, 2, \ldots n$ where n is the rank of `ARRAY`.

Examples. Assuming `A` is declared by the statement

```
INTEGER A(3:100, 200)
```

and is argument associated with `B`, the value of `GLOBAL_UBOUND(B)` is $\left[\begin{array}{cc} 100 & 200 \end{array}\right]$. If `B` is argument associated with the section, `A(5:10, 7:10)`, the value of `GLOBAL_UBOUND(B, 1)` is 6.

A.2.3.9 ABSTRACT_TO_PHYSICAL(ARRAY, INDEX, PROC)

Description. Returns processor identification for the physical processor associated with a specified abstract processor relative to a global actual argument array.

Class. Subroutine.

Arguments.

ARRAY	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.
INDEX	must be a rank-1 integer array containing the coordinates of an abstract processor in the processors arrangement onto which the global HPF array is mapped. It is an INTENT(IN) argument. The size of INDEX must equal the rank of the processors arrangement.
PROC	must be scalar and of type integer. It is an INTENT(OUT) argument. It receives the identifying value for the physical processor associated with the abstract processor specified by INDEX.

A.2.3.10 PHYSICAL_TO_ABSTRACT(ARRAY, PROC, INDEX)

Description. Returns coordinates for an abstract processor, relative to a global actual argument array, corresponding to a specified physical processor.

Class. Subroutine.

Arguments.

ARRAY	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an INTENT(IN) argument.
PROC	must be scalar and of type default integer. It is an INTENT(IN) argument. It contains an identifying value for a physical processor.
INDEX	must be a rank-1 integer array. It is an INTENT(OUT) argument. The size of INDEX must equal the rank of the processor arrangement onto which the global HPF array is mapped. INDEX receives the coordinates within this processors arrangement of the abstract processor associated with the physical processor specified by PROC.

This procedure can be used only on systems where there is a one-to-one correspondence between abstract processors and physical processors. On systems where this correspondence is one-to-many an equivalent, system-dependent procedure should be provided.

A.2.3.11 LOCAL_TO_GLOBAL(ARRAY, L_INDEX, G_INDEX)

Description. Converts a set of local coordinates within a local dummy array to an equivalent set of global coordinates within the associated global HPF actual argument array.

Class. Subroutine.

Arguments.

`ARRAY`	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an `INTENT(IN)` argument.
`L_INDEX`	must be a rank-1 integer array whose size is equal to the rank of `ARRAY`. It is an `INTENT(IN)` argument. It contains the coordinates of an element within the local dummy array `ARRAY`.
`G_INDEX`	must be a rank-1 integer array whose size is equal to the rank of `ARRAY`. It is an `INTENT(OUT)` argument. It receives the coordinates within the global HPF array actual argument of the element identified within the local array by `L_INDEX`.

A.2.3.12 GLOBAL_TO_LOCAL(ARRAY, G_INDEX, L_INDEX, LOCAL, NCOPIES, PROCS)

Optional arguments. `L_INDEX`, `LOCAL`, `NCOPIES`, `PROCS`

Description. Converts a set of global coordinates within a global HPF actual argument array to an equivalent set of local coordinates within the associated local dummy array.

Class. Subroutine.

Arguments.

`ARRAY`	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument. It is an `INTENT(IN)` argument.
`G_INDEX`	must be a rank-1 integer array whose size is equal to the rank of `ARRAY`. It is an `INTENT(IN)` argument. It contains the coordinates of an element within the global HPF array actual argument associated with the local dummy array `ARRAY`.
`L_INDEX` (optional)	must be a rank-1 integer array whose size is equal to the rank of `ARRAY`. It is an `INTENT(OUT)` argument. It receives the coordinates within a local dummy array of the element identified within the global actual argument array by `G_INDEX`. (These coordinates are identical on any processor that holds a copy of the identified global array element.) However, the values in `L_INDEX` are undefined if the value returned (or that would be returned) in `LOCAL` is false.

`LOCAL` (optional)	must be scalar and of type `LOGICAL`. It is an `INTENT(OUT)` argument. It is set to `.TRUE.` if the local array contains a copy of the global array element and to `.FALSE.` otherwise.
`NCOPIES` (optional)	must be scalar and of type integer. It is an `INTENT(OUT)` argument. It is set to the number of processors that hold a copy of the identified element of the global actual array.
`PROCS` (optional)	must be a rank-1 integer array whose size is at least the number of processors that hold copies of the identified element of the global actual array. The identifying numbers of those processors are placed in `PROCS`. The order in which they appear is implementation dependent.

A.2.4 MY_PROCESSOR()

Description. Returns the identifying number of the calling physical processor.

Class. Pure function.

Result Type, Type Parameter, and Shape. The result is scalar and of type default integer.

Result Value. Returns the identifying number of the physical processor from which the call is made. This value is in the range $0 \leq$ `MY_PROCESSOR` $\leq n-1$ where n is the value returned by `NUMBER_OF_PROCESSORS`.

A.2.5 LOCAL_BLKCNT(ARRAY, DIM, PROC)

Optional arguments. `DIM`, `PROC`.

Description. Returns the number of blocks of elements in each dimension, or of a specific dimension of the array on a given processor.

Class. Pure function.

Arguments.

`ARRAY`	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.
`DIM` (optional)	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`. The corresponding actual argument must not be an optional dummy argument.
`PROC` (optional)	must be scalar and of type integer. It must be a valid processor number.

Result Type, Type Parameter, and Shape. The result is of type default integer. It is scalar if `DIM` is present; otherwise the result is an array of rank one and size n, where n is the rank of `ARRAY`.

Result Value.

Case (i): The value of `LOCAL_BLKCNT(ARRAY, DIM, PROC)` is the number of blocks of the ultimate align target of `ARRAY` in dimension `DIM` that are mapped to processor `PROC` and which have at least one element of `ARRAY` aligned to them.

Case (ii): `LOCAL_BLKCNT(ARRAY, DIM)` returns the same value as `LOCAL_BLKCNT(ARRAY, DIM, PROC=MY_PROCESSOR())`.

Case (iii): `LOCAL_BLKCNT(ARRAY)` has a value whose ith component is equal to `LOCAL_BLKCNT(ARRAY,` i`)`, for $i = 1, \ldots, n$, where n is the rank of `ARRAY`.

Examples. Given the declarations

```
      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)
```

the values returned on the physical processor corresponding to `PR(2,4)` in `NBX` is $\begin{bmatrix} 4 & 3 \end{bmatrix}$ and in `NBY` is $\begin{bmatrix} 1 \end{bmatrix}$.

A.2.6 LOCAL_LINDEX(ARRAY, DIM, PROC)

Optional argument. `PROC`.

Description. Returns the lowest local index of all blocks of an array dummy argument in a given dimension on a processor.

Class. Pure function.

Arguments.

`ARRAY`	may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.
`DIM`	must be scalar and of type integer with a value in the range $1 \leq$ `DIM` $\leq n$, where n is the rank of `ARRAY`.

PROC (optional) must be scalar and of type integer. It must be a valid processor number.

Result Type, Type Parameter, and Shape. The result is a rank-one array of type default integer and size b, where b is the value returned by LOCAL_BLKCNT(ARRAY, DIM [, PROC])

Result Value.

Case (i): The value of LOCAL_LINDEX(ARRAY, DIM, PROC) has a value whose ith component is the local index of the first element of the ith block in dimension DIM of ARRAY on processor PROC.

Case (ii): LOCAL_LINDEX(ARRAY, DIM) returns the same value as LOCAL_LINDEX(ARRAY, DIM, PROC=MY_PROCESSOR()).

Examples. With the same declarations as in the example under LOCAL_BLKCNT, on the physical processor corresponding to PR(2,4) the value returned by LOCAL_LINDEX(X, DIM=1) is $\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$; the value of LOCAL_LINDEX(X, DIM=2) is $\begin{bmatrix} 1 & 3 & 4 \end{bmatrix}$.

A.2.7 LOCAL_UINDEX(ARRAY, DIM, PROC)

Optional argument. PROC.

Description. Returns the highest local index of all blocks of an array dummy argument in a given dimension on a processor.

Class. Pure function.

Arguments.

ARRAY may be of any type; it must be a dummy array that is associated with a global HPF array actual argument.

DIM must be scalar and of type integer with a value in the range $1 \leq$ DIM $\leq n$, where n is the rank of ARRAY.

PROC (optional) must be scalar and of type integer. It must be a valid processor number.

Result Type, Type Parameter, and Shape. The result is a rank-one array of type default integer and size b, where b is the value returned by LOCAL_BLKCNT(ARRAY, DIM [, PROC])

Result Value.

Case (i): The value of LOCAL_UINDEX(ARRAY, DIM, PROC) has a value whose ith component is the local index of the last element of the ith block in dimension DIM of ARRAY on processor PROC.

Case (ii): LOCAL_UINDEX(ARRAY, DIM) returns the same value as LOCAL_UINDEX(ARRAY, DIM, PROC=MY_PROCESSOR()).

Examples. With the same declarations as in the example under LOCAL_BLKCNT, on the physical processor corresponding to PR(2,4) the value returned by LOCAL_UINDEX(X, DIM=1) is $\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$; the value of LOCAL_UINDEX(X, DIM=2) is $\begin{bmatrix} 2 & 3 & 4 \end{bmatrix}$.

A.3 Local Routines Written in Fortran 90

The suggested interface to local SPMD routines written in Fortran 90 is the same as that for HPF local routines, with these few exceptions:

- Only Fortran 90 constructs should be used; it may not be possible to use extensions peculiar to HPF such as `FORALL` and the HPF library routines.
- It is recommended that Fortran 90 language processors to be used for this purpose be extended to support the HPF local distribution query routines `GLOBAL_ALIGNMENT`, `GLOBAL_TEMPLATE`, and `GLOBAL_DISTRIBUTION` as described in Section A.2.3. It is also recommended that these facilities be defined in a Fortran 90 module named `HPF_LOCAL_LIBRARY`.
- Assuming that the intent is to compile such routines with a non-HPF Fortran 90 compiler, the Fortran 90 program text should be in separate files rather than incorporated into HPF source code.
- The suggested extrinsic kind keyword for this calling interface is `F90_LOCAL`.

The restrictions listed in Section A.2.1 ought to apply as well to local routines written in Fortran 90.

A.3.1 Argument Association

If a dummy argument in the HPF explicit extrinsic interface is an array, then the corresponding dummy argument in the specification of the local procedure must be an array of the same rank, type, and type parameters. When the extrinsic procedure is invoked, the local dummy argument is associated with the local array that consists of the subgrid of the global array that is stored locally. This local array will be a valid Fortran 90 array.

If a dummy argument in the HPF explicit extrinsic interface is a scalar then the corresponding dummy argument of the local procedure must be a scalar of the same type. When the extrinsic procedure is invoked then the local procedure is passed an argument that consists of the local copy of the replicated scalar. This copy will be a valid Fortran 90 scalar.

If an HPF explicit extrinsic interface defines a function, then the local procedure should be a Fortran 90 function that returns a scalar of the same type and type parameters, or an array of the same rank, type, and type parameters, as the HPF extrinsic function. The value returned by each local invocation is the local part of the value returned by the HPF invocation.

A.4 Example HPF Extrinsic Procedures

The first example shows an `INTERFACE` block, call, and subroutine definition for matrix multiplication:

```
!    The NEWMATMULT routine computes C=A*B.  A copy of row A(I,*) and
!    column B(*,J) is broadcast to the processor that computes C(I,J)
!    before the call to NEWMATMULT.
```

```
      INTERFACE
        EXTRINSIC(HPF_LOCAL) SUBROUTINE NEWMATMULT(A, B, C)
          REAL,  DIMENSION(:,:), INTENT(IN)  ::  A, B
          REAL,  DIMENSION(:,:), INTENT(OUT) ::  C
!HPF$     ALIGN A(I,J) WITH *C(I,*)
!HPF$     ALIGN B(I,J) WITH *C(*,J)
        END SUBROUTINE NEWMATMULT
      END INTERFACE
      ...
      CALL NEWMATMULT(A,B,C)
      ...

! The Local Subroutine Definition:
!    Each processor is passed 3 arrays of rank 2.  Assume that the
!    global HPF arrays A,B and C have dimensions LxM, MxN and LxN,
!    respectively.  The local array CC is (a copy of) a rectangular
!    subarray of C. Let I1,I2,...,Ir and J1,J2,...,Js be,
!    respectively, the row and column indices of this subarray at a
!    processor.  Then AA is (a copy of) the subarray of A with row
!    indices I1,...,Ir and column indices 1,...,M; and BB is (a copy
!    of) the subarray of B with row indices 1,...,M and column
!    indices J1,...,Js.   C may be replicated, in which case copies
!    of C(I,J) will be consistently updated at various processors.

      EXTRINSIC(HPF_LOCAL) SUBROUTINE NEWMATMULT(AA, BB, CC)
      REAL,  DIMENSION(:,:), INTENT(IN)  ::  AA, BB
      REAL,  DIMENSION(:,:), INTENT(OUT) ::  CC
!HPF$ ALIGN AA(I,J) WITH *CC(I,*)
!HPF$ ALIGN BB(I,J) WITH *CC(*,J)
      INTEGER I,J

!     loop uses local indices

      DO I = LBOUND(CC,1), UBOUND(CC,1)
        DO J = LBOUND(CC,2), UBOUND(CC,2)
           CC(I,J) = DOT_PRODUCT(AA(I,:), BB(:,J))
        END DO
      END DO
      RETURN
      END
```

The second example shows an `INTERFACE` block, call, and subroutine definition for sum reduction:

```
!    The SREDUCE routine computes at each processor the sum of
!    the local elements of an array of rank 1.  It returns an
!    array that consists of one sum per processor.  The sum
!    reduction is completed by reducing this array of partial
```

```
 !    sums.  The function fails if the array is replicated.
 !    (Replicated arrays could be handled by a more complicated code.)

       INTERFACE
         EXTRINSIC(HPF_LOCAL) FUNCTION SREDUCE(A) RESULT(R)
           REAL, DIMENSION(NUMBER_OF_PROCESSORS()) :: R
 !HPF$      DISTRIBUTE (BLOCK) :: R
           REAL,  DIMENSION(:), INTENT(IN)  ::  A
         END FUNCTION SREDUCE
       END INTERFACE
       ...
       TOTAL = SUM(SREDUCE(A))
       ...

 ! The Local Subroutine Definition
       EXTRINSIC(HPF_LOCAL) FUNCTION SREDUCE(AA) RESULT(R)
       REAL, DIMENSION(1) :: R
 !HPF$ DISTRIBUTE (BLOCK) :: R
       REAL,  DIMENSION(:), INTENT(IN)  ::  AA

       INTEGER COPIES

       CALL GLOBAL_ALIGNMENT(AA, NUMBER_OF_COPIES = COPIES)
       IF (COPIES > 1) CALL ERROR()                ! array is replicated
 ! Additional code to check that template is not replicated
       ...
 ! Array is not replicated -- compute local sum
       R(1) = SUM(AA)
       RETURN
       END
```

The DISTRIBUTE directive in the local function SREDUCE specifies that the global actual argument is to have block distribution; the subarray seen on any particular processor during local execution will of course reside entirely within that processor.

Instead of including the interface block in the caller, one could also enclose the definition of SREDUCE in a module called, say, REDUCTION, and then replace the interface block with the statement

```
       USE REDUCTION
```

Annex B

Coding Single Processor Routines in HPF

This annex defines a set of conventions for writing code in which an instance of a subprogram executes on only one processor (of which there may be more than one).

If a program unit has extrinsic kind HPF_SERIAL, an HPF compiler should assume that the subprogram is coded to be executed on a single processor. From the point of view of a global HPF caller, the HPF_SERIAL procedure behaves the same as an identically coded HPF procedure would. Differences might only arise in implementation-specific behavior (such as the performance).

The EXTRINSIC mechanism, which allows an HPF programmer to declare a calling interface to a non-HPF subprogram, is described in Section 6 of the HPF specification.

B.1 Conventions for Uniprocessor Subprograms

The rules stated in section 14.7 of the Fortran 90 standard will apply to variables defined in HPF_SERIAL scoping units. In particular, if the definition status, association status, or allocation status of a variable is defined upon execution of a RETURN statement or an END statement in a Fortran 90 subprogram, such a variable in an HPF_SERIAL subprogram will be defined upon execution of a RETURN statement or an END statement.

As is the case with HPF_LOCAL, any I/O performed within an HPF_SERIAL subprogram, and the correspondence of file names and unit numbers used to those used in global HPF and HPF_LOCAL code will be implementation-defined.

B.1.1 Calling Sequence

Prior to invocation of an HPF_SERIAL procedure from global HPF, the behavior of the program will be as if the following actions take place:

1. The processors are synchronized. All actions that logically precede the call are completed.

2. All actual arguments are remapped to the processor that will actually execute the HPF_SERIAL procedure. The argument will appear to the HPF_SERIAL procedure as a sequential argument.

The behavior of the HPF_SERIAL procedure will be as if it was executed on only one processor. After the instance of the HPF_SERIAL procedure invoked from global HPF has completed, the behavior will be as if the following happen:

1. All processors are synchronized after the call.

2. The original mappings of actual arguments are restored.

B.2 Serial Routines Written in HPF

A subprogram may be defined to be of extrinsic kind HPF_SERIAL (and be compiled by an HPF compiler). The *subroutine-stmt* or *function-stmt* that begins the subprogram must contain the prefix EXTRINSIC(HPF_SERIAL).

B.2.1 Restrictions

There are restrictions that apply to an HPF_SERIAL subprogram.

No *specification-directive*, *realign-directive*, or *redistribute-directive* is permitted to be appear in an HPF_SERIAL subprogram or interface body.

> *Rationale.* An HPF mapping directive would likely be meaningless in an HPF_SERIAL subprogram. Note, however, that the *independent-directive* may appear in an HPF_SERIAL subprogram, since it may provide meaningful information to a compiler about a DO loop or a FORALL statement or construct. (*End of rationale.*)

Any dummy data objects and any function result variables of an HPF_SERIAL procedure will be considered to be sequential.

An HPF_SERIAL subprogram must not contain a definition of a common block that has the same name as a common block defined in an HPF or HPF_LOCAL program unit. In addition, an HPF_SERIAL subprogram must not contain a definition of the blank common block if an HPF or HPF_LOCAL program unit has a definition of the blank common block.

A dummy argument or function result variable of an HPF_SERIAL procedure that is referenced in global HPF must not have the POINTER attribute. A subobject of a dummy argument or function result of an HPF_SERIAL procedure that is referenced in global HPF, must not have the POINTER attribute.

A dummy argument of an HPF_SERIAL procedure that is referenced in global HPF and any subobject of such a dummy argument must not have the TARGET attribute.

A dummy procedure argument of an HPF_SERIAL procedure must be an HPF_SERIAL procedure.

An HPF_SERIAL procedure referenced in global HPF must have an accessible explicit interface.

An HPF_SERIAL subprogram must not contain a reference to a procedure that has *extrinsic-kind* HPF or HPF_LOCAL.

A reference to an HPF_SERIAL procedure must not appear in an HPF_LOCAL unit.

There is currently no manner in which to specify which processor is to execute an HPF_SERIAL procedure.

B.3 Intrinsic and Library Procedures

An HPF_SERIAL subprogram may contain references to any HPF intrinsic function or HPF_LIBRARY procedure, except HPF_ALIGNMENT, HPF_DISTRIBUTION or HPF_TEMPLATE. The HPF_LOCAL_LIBRARY module must not be used within an HPF_SERIAL scope.

References to the intrinsic functions NUMBER_OF_PROCESSORS and PROCESSORS_SHAPE will return the same value as if the function reference appeared in global HPF.

B.4 Example HPF_SERIAL Extrinsic Procedure

```
      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)

!  Compute DATA_ARRAY
        ...
        CALL DISPLAY_DATA(DATA_ARRAY)
      END PROGRAM MY_TEST

! The definition of a graphical display subroutine.  In some implementation-
! dependent fashion, this will plot a graph of the data in DATA.
      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
```

Annex C

Syntax Rules

C.2 High Performance Fortran Terms and Concepts

C.2.3 Syntax of Directives

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	*dynamic-directive*
		or	*inherit-directive*
		or	*template-directive*
		or	*combined-directive*
		or	*sequence-directive*
H205	*executable-directive*	**is**	*realign-directive*
		or	*redistribute-directive*
		or	*independent-directive*

Constraint: An *hpf-directive-line* cannot be commentary following another statement on the same line.

Constraint: A *specification-directive* may appear only where a *declaration-construct* may appear.

Constraint: An *executable-directive* may appear only where an *executable-construct* may appear.

Constraint: An *hpf-directive-line* follows the rules of either Fortran 90 free form (3.3.1.1) or fixed form (3.3.2.1) comment lines, depending on the source form of the surrounding Fortran 90 source form in that program unit. (3.3)

C.3 Data Alignment and Distribution Directives

C.3.2 Syntax of Data Alignment and Distribution Directives

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

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

Constraint: The same *combined-attribute* must not appear more than once in a given *combined-directive*.

Constraint: If the `DIMENSION` attribute appears in a *combined-directive*, any entity to which it applies must be declared with the HPF `TEMPLATE` or `PROCESSORS` type specifier.

C.3.3 DISTRIBUTE and REDISTRIBUTE Directives

H303 *distribute-directive* **is** `DISTRIBUTE` *distributee dist-directive-stuff*

H304 *redistribute-directive* **is** `REDISTRIBUTE` *distributee dist-directive-stuff*
or `REDISTRIBUTE` *dist-attribute-stuff* :: *distributee-list*

H305 *dist-directive-stuff* **is** *dist-format-clause* [*dist-onto-clause*]

H306 *dist-attribute-stuff* **is** *dist-directive-stuff*
or *dist-onto-clause*

H307 *distributee* **is** *object-name*
or *template-name*

H308 *dist-format-clause* **is** (*dist-format-list*)
or * (*dist-format-list*)
or *

H309 *dist-format* **is** `BLOCK` [(*int-expr*)]
or `CYCLIC` [(*int-expr*)]
or *

H310 *dist-onto-clause* **is** `ONTO` *dist-target*

H311 *dist-target* **is** *processors-name*
or * *processors-name*
or *

Constraint: An *object-name* mentioned as a *distributee* must be a simple name and not a subobject designator.

Constraint: An *object-name* mentioned as a *distributee* may not appear as an *alignee*.

Constraint: An *object-name* mentioned as a *distributee* may not have the `POINTER` attribute.

Constraint: A *distributee* that appears in a `REDISTRIBUTE` directive must have the `DYNAMIC` attribute (see Section 3.5).

Constraint: If a *dist-format-list* is specified, its length must equal the rank of each *distributee*.

Constraint: If both a *dist-format-list* and a *processors-name* appear, the number of elements of the *dist-format-list* that are not "*" must equal the rank of the named processor arrangement.

Constraint: If a *processors-name* appears but not a *dist-format-list*, the rank of each *distributee* must equal the rank of the named processor arrangement.

Constraint: If either the *dist-format-clause* or the *dist-target* in a `DISTRIBUTE` directive begins with "*" then every *distributee* must be a dummy argument.

Constraint: Neither the *dist-format-clause* nor the *dist-target* in a `REDISTRIBUTE` may begin with "*".

Constraint: Any *int-expr* appearing in a *dist-format* of a `DISTRIBUTE` directive must be a *specification-expr*.

C.3.4 ALIGN and REALIGN Directives

H312	*align-directive*	**is**	`ALIGN` *alignee* *align-directive-stuff*
H313	*realign-directive*	**is**	`REALIGN` *alignee* *align-directive-stuff*
		or	`REALIGN` *align-attribute-stuff* `::` *alignee-list*
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*

Constraint: An *object-name* mentioned as an *alignee* must be a simple name and not a subobject designator.

Constraint: An *object-name* mentioned as an *alignee* may not appear as a *distributee*.

Constraint: An *object-name* mentioned as an *alignee* may not have the `POINTER` attribute.

Constraint: Any *alignee* that appears in a `REALIGN` directive must have the `DYNAMIC` attribute (see Section 3.5).

Constraint: If the `align-target` specified in the `align-with-clause` has the `DYNAMIC` attribute, then each `alignee` must also have the `DYNAMIC` attribute.

Constraint: If the *alignee* is scalar, the *align-source-list* (and its surrounding parentheses) must not appear. In this case the statement form of the directive is not allowed.

Constraint: If the *align-source-list* is present, its length must equal the rank of the alignee.

Constraint: An *align-dummy* must be a named variable.

Constraint: An object may not have both the `INHERIT` attribute and the `ALIGN` attribute. (However, an object with the `INHERIT` attribute may appear as an *alignee* in a `REALIGN` directive, provided that it does not appear as a *distributee* in a `DISTRIBUTE` or `REDISTRIBUTE` directive.)

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*

Constraint: An *object-name* mentioned as an *align-target* must be a simple name and not a subobject designator.

Constraint: An *align-target* may not have the `OPTIONAL` attribute.

Constraint: If the *align-spec* in an `ALIGN` directive begins with "*" then every *alignee* must be a dummy argument.

Constraint: The *align-spec* in a `REALIGN` may not begin with "*".

Constraint: Each *align-dummy* may appear at most once in an *align-subscript-list*.

Constraint: An *align-subscript-use* expression may contain at most one occurrence of an *align-dummy*.

Constraint: An *align-dummy* may not appear anywhere in the *align-spec* except where explicitly permitted to appear by virtue of the grammar shown above. Paraphrased, one may construct an *align-subscript-use* by starting with an *align-dummy* and then doing additive and multiplicative things to it with any integer expressions that contain no *align-dummy*.

Constraint: A *subscript* in an *align-subscript* may not contain occurrences of any *align-dummy*.

Constraint: An *int-add-operand*, *int-mult-operand*, or *int-level-two-expr* must be of type integer.

C.3.5 DYNAMIC Directive

H329 *dynamic-directive* **is** `DYNAMIC` *alignee-or-distributee-list*

H330 *alignee-or-distributee* **is** *alignee*
or *distributee*

Constraint: An object in `COMMON` may not be declared `DYNAMIC` and may not be aligned to an object (or template) that is `DYNAMIC`. (To get this kind of effect, Fortran 90 modules must be used instead of `COMMON` blocks.)

Constraint: An object with the `SAVE` attribute may not be declared `DYNAMIC` and may not be aligned to an object (or template) that is `DYNAMIC`.

C.3.7 PROCESSORS Directive

H331 *processors-directive* **is** `PROCESSORS` *processors-decl-list*

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

H333 *processors-name* **is** *object-name*

C.3.8 TEMPLATE Directive

H334 *template-directive* **is** `TEMPLATE` *template-decl-list*

H335 *template-decl* **is** *template-name* [(*explicit-shape-spec-list*)]

H336 *template-name* **is** *object-name*

C.3.9 INHERIT Directive

H337 *inherit-directive* **is** `INHERIT` *dummy-argument-name-list*

C.4 Data Parallel Statements and Directives

C.4.1 The FORALL Statement

H401 *forall-stmt* **is** `FORALL` *forall-header forall-assignment*

H402 *forall-header* **is** (*forall-triplet-spec-list* [, *scalar-mask-expr*])

Constraint: Any procedure referenced in the *scalar-mask-expr* of a *forall-header* must be pure, as defined in Section 4.3.

H403 *forall-triplet-spec* **is** *index-name* = *subscript* : *subscript* [: *stride*]

Constraint: *index-name* must be a scalar integer variable.

Constraint: A *subscript* or *stride* in a *forall-triplet-spec-list* must not contain a reference to any *index-name* in the *forall-triplet-spec-list* in which it appears.

H404 *forall-assignment* **is** *assignment-stmt*
or *pointer-assignment-stmt*

Constraint: Any procedure referenced in a *forall-assignment*, including one referenced by a defined operation or assignment, must be pure as defined in Section 4.3.

C.4.2 The FORALL Construct

H405 *forall-construct* **is** `FORALL` *forall-header*
forall-body-stmt
[*forall-body-stmt*] ...
`END FORALL`

H406 *forall-body-stmt* **is** *forall-assignment*
or *where-stmt*
or *where-construct*
or *forall-stmt*
or *forall-construct*

Constraint: Any procedure referenced in a *forall-body-stmt*, including one referenced by a defined operation or assignment, must be pure as defined in Section 4.3.

Constraint: If a *forall-stmt* or *forall-construct* is nested in a *forall-construct*, then the inner `FORALL` may not redefine any *index-name* used in the outer *forall-construct*.

C.4.3 Pure Procedures

H407 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

H408 *prefix-spec* **is** *type-spec*
or `RECURSIVE`
or `PURE`
or *extrinsic-prefix*

H409 *function-stmt* **is** [*prefix*] `FUNCTION` *function-name function-stuff*

H410 *function-stuff* **is** ([*dummy-arg-name-list*]) [`RESULT` (*result-name*)]

H411 *subroutine-stmt* **is** [*prefix*] `SUBROUTINE` *subroutine-name subroutine-stuff*

H412 *subroutine-stuff* **is** [([*dummy-arg-list*])]

Constraint: A *prefix* must contain at most one of each variety of *prefix-spec*.

Constraint: The *prefix* of a *subroutine-stmt* must not contain a *type-spec*.

Constraint: The *specification-part* of a pure function must specify that all dummy arguments have `INTENT(IN)` except procedure arguments and arguments with the `POINTER` attribute.

Constraint: A local variable declared in the *specification-part* or *internal-subprogram-part* of a pure function must not have the `SAVE` attribute.

> *Advice to users.* Note local variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the `SAVE` attribute; therefore, such initialization is also disallowed. (*End of advice to users.*)

Constraint: The *execution-part* and *internal-subprogram-part* of a pure function may not use a dummy argument, a global variable, or an object that is storage associated with a global variable, or a subobject thereof, in the following contexts:

- As the assignment variable of an *assignment-stmt*;
- As a `DO` variable or implied `DO` variable, or as an *index-name* in a *forall-triplet-spec*;
- As an *input-item* in a *read-stmt*;
- As an *internal-file-unit* in a *write-stmt*;
- As an `IOSTAT=` or `SIZE=` specifier in an I/O statement.
- In an *assign-stmt*;
- As the *pointer-object* or *target* of a *pointer-assignment-stmt*;
- As the *expr* of an *assignment-stmt* whose assignment variable is of a derived type, or is a pointer to a derived type, that has a pointer component at any level of component selection;
- As an *allocate-object* or *stat-variable* in an *allocate-stmt* or *deallocate-stmt*, or as a *pointer-object* in a *nullify-stmt*; or
- As an actual argument associated with a dummy argument with `INTENT (OUT)` or `INTENT(INOUT)` or with the `POINTER` attribute.

Constraint: Any procedure referenced in a pure function, including one referenced via a defined operation or assignment, must be pure.

Constraint: A dummy argument or the dummy result of a pure function may be explicitly aligned only with another dummy argument or the dummy result, and may not be explicitly distributed or given the `INHERIT` attribute.

Constraint: In a pure function, a local variable may be explicitly aligned only with another local variable, a dummy argument, or the result variable. A local variable may not be explicitly distributed.

Constraint: In a pure function, a dummy argument, local variable, or the result variable must not have the `DYNAMIC` attribute.

Constraint: In a pure function, a global variable must not appear in a *realign-directive* or *redistribute-directive*.

Constraint: A pure function must not contain a *backspace-stmt*, *close-stmt*, *endfile-stmt*, *inquire-stmt*, *open-stmt*, *print-stmt*, *rewind-stmt*, or a *read-stmt* or *write-stmt* whose *io-unit* is an *external-file-unit* or `*`.

Constraint: A pure function must not contain a *pause-stmt* or *stop-stmt*.

Constraint: The *specification-part* of a pure subroutine must specify the intents of all dummy arguments except procedure arguments and arguments that have the `POINTER` attribute.

Constraint: A local variable declared in the *specification-part* or *internal-function-part* of a pure subroutine must not have the `SAVE` attribute.

Constraint: The *execution-part* or *internal-subprogram-part* of a pure subroutine must not use a dummy parameter with `INTENT(IN)`, a global variable, or an object that is storage associated with a global variable, or a subobject thereof, in the following contexts:

- As the assignment variable of an *assignment-stmt*;
- As a `DO` variable or implied `DO` variable, or as a *index-name* in a *forall-triplet-spec*;
- As an *input-item* in a *read-stmt*;
- As an *internal-file-unit* in a *write-stmt*;
- As an `IOSTAT=` or `SIZE=` specifier in an I/O statement.
- In an *assign-stmt*;
- As the *pointer-object* or *target* of a *pointer-assignment-stmt*;
- As the *expr* of an *assignment-stmt* whose assignment variable is of a derived type, or is a pointer to a derived type, that has a pointer component at any level of component selection;
- As an *allocate-object* or *stat-variable* in an *allocate-stmt* or *deallocate-stmt*, or as a *pointer-object* in a *nullify-stmt*;
- As an actual argument associated with a dummy argument with `INTENT (OUT)` or `INTENT(INOUT)` or with the `POINTER` attribute.

Constraint: Any procedure referenced in a pure subroutine, including one referenced via a defined operation or assignment, must be pure.

Constraint: A dummy argument of a pure subroutine may be explicitly aligned only with another dummy argument, and may not be explicitly distributed or given the `INHERIT` attribute.

Constraint: In a pure subroutine, a local variable may be explicitly aligned only with another local variable or a dummy argument. A local variable may not be explicitly distributed.

Constraint: In a pure subroutine, a dummy argument or local variable must not have the `DYNAMIC` attribute.

Constraint: In a pure subroutine, a global variable must not appear in a *realign-directive* or *redistribute-directive*.

Constraint: A pure subroutine must not contain a *backspace-stmt*, *close-stmt*, *endfile-stmt*, *inquire-stmt*, *open-stmt*, *print-stmt*, *rewind-stmt*, *print-stmt*, or a *read-stmt* or *write-stmt* whose *io-unit* is an *external-file-unit* or `*`.

Constraint: A pure subroutine must not contain a *pause-stmt* or *stop-stmt*.

Constraint: A pure subroutine must not contain an asterisk (*) in its *dummy-argument-list*.

Constraint: An *interface-body* of a pure procedure must specify the intents of all dummy arguments except `POINTER` and procedure arguments.

Constraint: In a reference to a pure procedure, a *procedure-name actual-arg* must be the name of a pure procedure.

C.4.4 The INDEPENDENT Directive

H413	*independent-directive*	**is**	`INDEPENDENT` [, *new-clause*]
H414	*new-clause*	**is**	`NEW` (*variable-list*)

Constraint: The first non-comment line following an *independent-directive* must be a *do-stmt*, *forall-stmt*, or a *forall-construct*.

Constraint: If the first non-comment line following an *independent-directive* is a *do-stmt*, then that statement must contain a *loop-control* option containing a *do-variable*.

Constraint: If the `NEW` option is present, then the directive must apply to a `DO` loop.

Constraint: A *variable* named in the `NEW` option or any component or element thereof must not:

- Be a pointer or dummy argument; nor
- Have the `SAVE` or `TARGET` attribute.

C.6 Extrinsic Procedures

C.6.2 Definition and Invocation of Extrinsic Procedures

H601	*extrinsic-prefix*	**is**	`EXTRINSIC` (*extrinsic-kind-keyword*)
H602	*extrinsic-kind-keyword*	**is**	`HPF`
		or	`HPF_LOCAL`
		or	`HPF_SERIAL`
H603	*program-stmt*	**is**	[*extrinsic-prefix*] `PROGRAM` *program-name*
H604	*module-stmt*	**is**	[*extrinsic-prefix*] `MODULE` *module-name*
H605	*block-data-stmt*	**is**	[*extrinsic-prefix*] `BLOCK DATA` *block-data-name*

C.7 Storage and Sequence Association

C.7.1 Storage Association

H701	*sequence-directive*	**is**	`SEQUENCE` [[::] *association-name-list*]
		or	`NO SEQUENCE` [[::] *association-name-list*]

H702	*association-name*	**is**	*object-name*
		or	*function-name*
		or	`/` [*common-block-name*] `/`

Constraint: A variable or `COMMON` block name may appear at most once in a *sequence-directive* within any scoping unit.

Constraint: Only one sequence directive with a given *association-name* is permitted in the same scoping unit.

Annex D

Syntax Cross-reference

D.1 Nonterminal Symbols That Are Defined

Symbol	Defined	Referenced
add-op	R710	H323
add-operand	R706	H326
align-add-operand	H324	H323 H324
align-attribute-stuff	H315	H302 H313
align-directive	H312	H204
align-directive-stuff	H314	H312 H313
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-with-clause	H319	H314 H315
alignee	H316	H312 H313 H330
alignee-or-distributee	H330	H329
allocate-object	R625	
allocate-stmt	R622	
array-constructor	R431	
array-spec	R512	
assign-stmt	R838	
assignment-stmt	R735	H404
association-name	H702	H701
block-data-stmt	H605	
call-stmt	R1210	
combined-attribute	H302	H301
combined-directive	H301	H204
data-stmt	R529	
deallocate-stmt	R631	
directive-origin	H202	H201
dist-attribute-stuff	H306	H302 H304
dist-directive-stuff	H305	H303 H304 H306

dist-format	H309	H308
dist-format-clause	H308	H305
dist-onto-clause	H310	H305 H306
dist-target	H311	H310
distribute-directive	H303	H204
distributee	H307	H303 H304 H330
dummy-arg	R1221	H412
dynamic-directive	H329	H204
end-function-stmt	R1218	
end-subroutine-stmt	R1222	
entity-decl	R504	H301
executable-construct	R215	
executable-directive	H205	H203
execution-part	R208	
explicit-shape-spec	R513	H302 H332 H335
expr	R723	
extrinsic-kind-keyword	H602	H601
extrinsic-prefix	H601	H408 H603 H604 H605
forall-assignment	H404	H401 H406
forall-body-stmt	H406	H405
forall-construct	H405	H406
forall-header	H402	H401 H405
forall-stmt	H401	H406
forall-triplet-spec	H403	H402
function-reference	R1209	
function-stmt	H409	
function-stuff	H410	H409
function-subprogram	R1215	
hpf-directive	H203	H201
hpf-directive-line	H201	
independent-directive	H413	H205
inherit-directive	H337	H204
input-item	R914	
int-add-operand	H326	H323 H324
int-expr	R728	H309 H322
int-level-two-expr	H328	H323
int-mult-operand	H327	H324
int-variable	R607	H318
interface-body	R1204	
internal-subprogram-part	R210	
level-2-expr	R707	H328
mask-expr	R741	H402
module-stmt	H604	
mult-operand	R705	H327
namelist-group-object	R737	
namelist-stmt	R543	
new-clause	H414	H413
nullify-stmt	R629	

output-item	R915	
pause-stmt	R844	
pointer-assignment-stmt	R736	H404
pointer-object	R630	
prefix	H407	H409 H411
prefix-spec	H408	H407
processors-decl	H332	H331
processors-directive	H331	H204
processors-name	H333	H311 H332
program-stmt	H603	
read-stmt	R737	
realign-directive	H313	H205
redistribute-directive	H304	H205
section-subscript	R618	
sequence-directive	H701	H204
specification-directive	H204	H203
specification-expr	R734	
specification-part	R204	
stat-variable	R623	
stop-stmt	R842	
stride	R620	H403
subroutine-stmt	H411	
subroutine-stuff	H412	H411
subscript	R617	H403
subscript-triplet	R619	H322
target	R737	
template-decl	H335	H334
template-directive	H334	H204
template-name	H336	H307 H321 H335
type-declaration-stmt	R501	
type-spec	R502	H408
variable	R601	H414
where-construct	R739	H406
where-stmt	R738	H406
write-stmt	R737	

D.2 Nonterminal Symbols That Are Not Defined

Symbol	**Referenced**
block-data-name	H605
common-block-name	H702
dummy-arg-name	H410
dummy-argument-name	H337
function-name	H409 H702
index-name	H403
module-name	H604
object-name	H307 H316 H321 H333 H336 H702

program-name	H603
result-name	H410
subroutine-name	H411

D.3 Terminal Symbols

Symbol	**Referenced**
`!HPF$`	H202
`(`	H302 H308 H309 H314 H315 H320 H325 H332 H335 H402 H410 H412 H414 H601
`)`	H302 H308 H309 H314 H315 H320 H325 H332 H335 H402 H410 H412 H414 H601
`*`	H308 H309 H311 H317 H320 H322 H324
`*HPF$`	H202
`,`	H402 H413
`/`	H702
`:`	H317 H403
`::`	H301 H304 H313 H701
`=`	H403
`ALIGN`	H302 H312
`BLOCK`	H309 H605
`CHPF$`	H202
`CYCLIC`	H309
`DATA`	H605
`DIMENSION`	H302
`DISTRIBUTE`	H302 H303
`DYNAMIC`	H302 H329
`END`	H405
`EXTRINSIC`	H601
`FORALL`	H401 H405
`FUNCTION`	H409
`HPF`	H602
`HPF_LOCAL`	H602
`HPF_SERIAL`	H602
`INDEPENDENT`	H413
`INHERIT`	H302 H337
`MODULE`	H604
`NEW`	H414
`NO`	H701
`ONTO`	H310
`PROCESSORS`	H302 H331
`PROGRAM`	H603
`PURE`	H408
`REALIGN`	H313
`RECURSIVE`	H408
`REDISTRIBUTE`	H304
`RESULT`	H410

SEQUENCE	H701
SUBROUTINE	H411
TEMPLATE	H302 H334
WITH	H319

Bibliography

[1] Jeanne C Adams, Walter S. Brainerd, Jeanne T. Martin, Brian T. Smith, and Jerrold L. Wagener. *Fortran 90 Handbook*, Intertext-McGraw Hill, 1992.

[2] Eugene Albert, Joan D. Lukas, and Guy L. Steele, Jr. "Data Parallel Computers and the FORALL Statement", *Journal of Parallel and Distributed Computing*, October 1991.

[3] American National Standards Institute, Inc., 1430 Broadway, New York, NY. *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, approved April 3, 1978.

[4] 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.

[5] Bernstein, A. J. "Analysis of Programs for Parallel Processing", *IEEE Transactions on Electronic Computers*, Vol. 15, pp 757-762, 1966.

[6] P. Brezany, M. Gerndt, P. Mehrotra and H. Zima. *Concurrent File Operations in a High Performance Fortran*

[7] Barbara Chapman, Piyush Mehrotra, and Hans Zima. *Programming in Vienna Fortran*, Scientific Programming 1,1, August 1992, Also published as: ACPC/TR 92-3, Austrian Center of Parallel Computation, March 1992.

[8] M. Chen and J. Wu. *Optimizing Fortran 90 Programs for Data Motion on Massively Parallel Systems*, Yale University, YALEU/DCS/TR-882, New Haven, CT, 1991.

[9] M. Chen and J. Cowie. "Prototyping Fortran 90 Compilers for Massively Parallel Machines", *SIGPLAN92*, 1992.

[10] Digital Equipment Corporation, Maynard, Massachusetts. *DECmpp 12000 Sx - High Performance Fortran Reference Manual*, February, 1993, [AA-PMAHC-TE].

[11] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. *Fortran D Language Specification*. Report COMP TR90-141 (Rice) and SCCS-42c (Syracuse), Department of Computer Science, Rice University, Houston, Texas, and Syracuse Center for Computational Science, Syracuse University, Syracuse, New York, April 1991.

[12] ISO. *Fortran 90*, May 1991. [ISO/IEC 1539: 1991 (E) and now ANSI X3.198-1992].

[13] 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.

[14] 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).

[15] High Performance Fortran Forum. *High Performance Fortran Journal of Development* CRPC-TR93300, Center for Research on Parallel Computation, Rice University, Houston, TX, May. 1993.

[16] C. Koelbel and D. Loveman and R. Schreiber and G. Steele, Jr. adn M. Zosel. *The HIgh Performance Fortran Handbook* MIT Press, Cambridge, MA, 1994.

[17] C. Koelbel and P. Mehrotra. "An Overview of High Performance Fortran", *Fortran Forum*, Vol. 11, No. 4, December, 1992.

[18] David B. Loveman. "High Performance Fortran", *IEEE Parallel & Distributed Technology*, Vol. 1, No. 1, February 1993.

[19] David B. Loveman. "Element Array Assignment - the FORALL Statement", *Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 6-9, 1992.

[20] MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, California. *MasPar Fortran Reference Manual*, May 1991. [Software Version 1.1, 9303-0000, Rev. A2].

[21] Piyush Mehrotra and J. Van Rosendale. "Programming Distributed Memory Architectures Using Kali", In: Nicolau,A. et al.(Eds): *Advances in Languages and Compilers for Parallel Processing*, pp.364-384, Pitman/MIT-Press, 1991.

[22] Andrew Meltzer, Douglas M. Pase, and Tom MacDonald. *Basic Features of the MPP Fortran Programming Model*, Cray Research, Inc, Eagan, Minnesota, August 19, 1992.

[23] John Merlin. *Techniques for the Automatic Parallelisation of 'Distributed Fortran 90'*, Technical Report SNARC 92-02, Dept. of Electronics and Comp. Science, Univ. of Southampton, November 1991.

[24] Michael Metcalf and John Reid. *Fortran 90 Explained*, Oxford University Press, 1990.

[25] Robert E. Millstein. "Control Structures in ILLIAC IV Fortran", *Communications of the ACM*, 16(10):621-627, October 1973.

[26] Douglas M. Pase, Tom MacDonald, and Andrew Meltzer. *MPP Fortran Programming Model*, Cray Research, Inc, Eagan, Minnesota, August 26, 1992.

[27] Guy L. Steele Jr. "High Performance Fortran: Status Report", em Workshop on Languages, Compilers, and Runtime Environments for Distributed-Memory Multiprocessors, *ACM SIGPlan Notices*, Vol. 28, No. 1, January 1993.

[28] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran Reference Manual*, July 1991.

[29] US Department of Defense. *Military Standard, MIL-STD-1753: FORTRAN, DoD Supplement to American National Standard X3.9-1978*, November 9, 1978.

[30] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. *Vienna Fortran - a Language Specification*, ICASE Interim Report 21, ICASE NASA Langley Research Center, Hampton, Virginia 23665, March 1992.