DRAFT
High Performance Fortran Language Specification

High Performance Fortran Forum

CRPC-TR92225
January 1993

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

D R A F T

High Performance Fortran Language Specification

High Performance Fortran Forum

January 25, 1993
Version 1.0 DRAFT

The High Performance Fortran Forum (HPFF), with participation from over 40 organizations, has been meeting since March 1992 to discuss and define a set of extensions to Fortran called High Performance Fortran (HPF). Our goal is 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.

This is a draft of what will become the Final Report, Version 1.0, of the High Performance Fortran Forum. This document contains all the technical features proposed for the language. This copy of the draft was processed by LaTeX on January 26, 1993.

HPFF invites comments on the technical content of HPF, as well as on the editorial presentation in the document. Comments received before March 1, 1993 will be considered in producing the final draft of Version 1.0 of the HPF Language Specification.

Please send comments by email to hpff-comments@cs.rice.edu. To facilitate the processing of comments we request that separate comment messages be submitted for each chapter of the document and that the chapter be clearly identified in the "Subject:" line of the message. Comments about general overall impressions of the HPF document should be labeled as Chapter 1. All comments on the language specification become the property of Rice University.

If email access is impossible for comment responses, hard copy may be sent to

```
HPF Comments
c/o Theresa Chatman
CITI/CRPC, Box 1892
Rice University
Houston, TX 77251
```

HPFF plans to process the feedback received at a meeting in March. Best efforts will be made to reply to comments submitted.

The following changes have been made in this draft since the publication of Version 0.4 at Supercomputing '92:

Chapter 2 The first paragraph of section 2.3 (about Fortran 90 notation) was moved to section 1.5.

Clarification that HPF directives may not be trailing commentary, however HPF may have trailing commentary.

Syntactic catagories were changed and constraints were added to specify where HPF directives may occur in a scoping unit.

Changed name to be "view-directive" for consistency.

Chapter 3 Reorganized and rewritten for clarity. INHERIT added.

Chapter 4 BNF symbol subscript-name is now called index-name.

The constraint: "The variable of an assignment-stmt must be a distinct object for each active combination of subscript-name values. In this context, two objects are considered distinct if they have no subobjects in common." has been moved to the interpretation section, as it is not statically checkable. A similar change has been made for the pointer-target constraint of the same flavor.

The set of examples has been significantly enlarged.

Scalarizations have been made more clearly into pseudocode, and explanations to some difficult-to-handle cases added. Some of the scalarizations have also been simplified (with the general case explained in text).

The WHERE construct is now allowed within a FORALL construct (the BNF previously omitted this case).

Pure procedures may now have dummy arguments with explicit distributions, if those distributions are inherited from the caller.

Chapter 5 Changed the names of the new reductions AND, OR, and EOR to IALL, IANY, and IPARITY.

Fixed a bug in the GRADE_UP example.

Fixed various stylistic problems.

Moved the mapping inquiry subsection to the intrinsics section, out of the library section.

Chapter 9 Clarified the status of the character array language to be not in the subset, and as a result, removed the character array intrinsics.

Noted that the HPF_LIB module is not part of the subset, along with the HPF library.

Only very restricted forms of alignment subscript expressions (of the form $m * i + n$ where m and n are integer expressions) are part of the subset.

Appendix A No changes.

Bibliography Correctly spell "Mehrotra" and "Gerndt". Add Metcalf and Reid.

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.

The technical development was carried out by subgroups, whose work was reviewed by the full committee. During the period of development of High Performance Fortran (HPF), 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 subsequently led a group to develop benchmarks for HPF. In addition, Clemens-August Thole organized a complementary group in Europe and was instrumental in making this an international

effort. Charles Koelbel took notes during every meeting and produced detailed minutes, including summaries of the discussions, that were invaluable to the subgroup heads in preparing successive revisions to the draft proposal.

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 January HPFF meeting, for which there is no accurate attendee list):

Alliant Computer Systems Corporation	David Reese
Amoco Production Company	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
DEC Massively Parallel Systems Group	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 Vienna	Barbara Chapman, Hans Zima
Yale University	Marina Chen, Aloke Majumdar

Many people contributed chapters or major sections to the final language specification, 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 difficult, if not impossible, to identify all of those who contributed to the discussions; the entire mailing list was well over 500 names long. The following list includes some of the active participants in the HPFF process not mentioned above:

N Arunasalam	Marc Baber	Babak Bagheri
Jason Behm	Peter Belmont	Mike Bernhardt
Keith Bierman	John Bolstad	William Camp
Duane Carbon	Richard Carpenter	Brice Cassenti
Doreen Cheng	Mark Christon	Fabien Coelho
Robert Corbett	Bill Crutchfield	James Demmel
J. C. Diaz	Alan Egolf	Bo Einarsson
Robert Ferrell	Geoffrey Fox	Rhys Francis
Hans-Hermann Frese	Steve Goldhaber	Brent Gorda
Rick Gorton	Robert Halstead	Reinhard von Hanxleden
Carol Hoover	Alan Karp	Anthony Kimball
Ross Knippe	Bruce Knobe	David Kotz
Tom Lake	Bryan Lawver	Bruce Leasure
Stewart Levin	David Levine	Theodore Lewis
Woody Lichtenstein	Kevin Robert Lind	Ruth Lovely
Doug MacDonald	Philippe Marquet	Jeanne Martin
Oliver McBryan	John Merlin	Michael Metcalf
Charles Mosher	Lenore Mullin	Yoichi Muraoka
Bernie Murray	Vicki Newton	Dale Nielsen
Jeff Painter	Cherri Pancake	John Reid
Harvey Richardson	Bob Riley	Ron Schmucker
Doug Scofield	David Serafini	Anthony Skjellum
Niraj Srivastava	Paul St. Pierre	Nick Stanford
Mia Stephens	Jaspal Subhlok	Hanna Szoke
Bernard Tourancheau	Alex Vasilevsky	Arthur Veen
Brian Wake	Karen Warren	D. C. B. Watson
Matthijs van Waveren	Robert Weaver	Fred Webb
Stephen Mark Whitley	Michael Wolfe	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, 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 confe·ences budget of the Center for Research on Parallel Computation, an NSF Science and T·chnology 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).

Chapter 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." 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 deemed 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 HPF language features fall into four categories with respect to Fortran 90:

- New directives;
- New language syntax;
- Library routines; and
- Language 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, namely the **FORALL** statement and certain 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 performace computing including additional reduction functions, combining scatter functions, prefix and suffix functions, and sorting functions.

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;
- Parallel statements;
- Extended intrinsic functions and standard library;
- EXTRINSIC procedures;
- Parallel i/o statements; and
- Changes in sequence and storage association.

In addition, a subset of HPF suitable for earlier implementation is defined.

1.3.1 Data Distribution Features

Modern parallel and sequential architectures attain their fastest speed when the data accessed exhibits locality of reference. Often, the sequential storage order implied by FORTRAN 77 and Fortran 90 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 (`DISTRIBUTE`). Compilers may interpret these annotations to improve storage allocation for data, subject to the constraint that semantically every data object has only 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. Chapter 3 describes the distribution features.

1.3.2 Parallel Statements

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. Chapter 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 intrinsics and as standard library functions. In addition, several system inquiry functions useful for controlling parallel execution are provided in HPF. Chapter 5 describes these functions and subroutines.

1.3.4 Extrinsic Pro dures

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. Chapter 6 describes this interface and its use.

1.3.5 Parallel I/O Statements

By a narrow vote, explicitly parallel I/O statements were excluded from HPF. There were several reasons for this, including the possibility of providing operating system support for parallel files, the lack of a clearly portable paradigm for parallel I/O, and lack of implementation experience. In making this decision, the committee expressed the hope that a follow-on effort would add I/O features later. Chapter 7 gives more details on the decision and its rationale. Section A.7 in the Journal of Development (at the end of this document) details some of the alternatives that were considered.

1.3.6 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. Chapter 8 describes the restrictions and a directive related to storage and sequence association.

1.4 Fortran 90 and HPF Subset

An important goal for HPF is early compiler availability. In recognition of the facts that full Fortran 90 compilers may not be available in a timely fashion on all platforms and that implementation of some of the HPF extensions proposed is more complex than for others, a formal HPF subset has been defined. 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, 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. Chapter 9 describes the HPF subset.

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 proposed language features are given in the style used in the Fortran 90 standard. 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. References in parentheses in the text refer to the Fortran 90 standard.

1.6 Organization of this Document

Chapter 1, this chapter, presents an overview of HPF.

Chapter 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.

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

- The distribution model;
- Features for aligning array elements which are accessed together;
- Features for distributing array elements among processors; and
- Features for ALLOCATABLE arrays and pointers.

Chapter 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.

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

- New computational intrinsic functions and extensions to existing intrinsic functions;
- Inquiry intrinsic functions to check system and data partitioning status; and
- A standard library of computational functions.

Chapter 6 describes extrinsic procedures in HPF. This includes:

- The extrinsic procedure interface; and
- A Fortran 90 binding for the extrinsic interface.

Chapter 7 describes Input/Output in HPF. Its primary focus is explaining why HPF does *not* extend Fortran 90 I/O.

Chapter 8 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.

Chapter 9 describes subset HPF, which may be implemented more quickly than full HPF. This includes:

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

Appendix A, the Journal of Development, describes several features that were considered but not accepted into HPF. In many of these cases, features were rejected for lack of time or consensus rather than because of technical flaws. We offer them to future language designers for consideration.

The Bibliography provides references to various HPF sources:

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

Chapter 2

High Performance Fortran Terms and Concepts

This chapter 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 computations in Fortran 90 will make it the programming language of choice for 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 [Fortran 90] 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.

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.

- Modules. The module facilities of Fortran 90 enable the practice of design implementation using data abstractions. 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 informa specified to provide better checking and optimization at the interface to other su ograms.

- 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 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 full Fortran 90 with all of the approved 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 sufficient 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 "parallel" constructs:

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

- The extrinsic procedure mechanism.

The above features allow the explicit user specification of a high degree of potential parallelism in a machine-independent fashion. In addition, extrinsic procedures provide an escape mechanism in HPF to allow the use of efficient machine-specific primitives by explicitly executing on a set number of processors.

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 data communication that occurs during the execution of an HPF program is partially determined by the HPF data distribution directives in Chapter 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 then communication is required to bring them together. The examples below illustrate how this may occur.

Clearly, there is a tradeoff between parallelism and communication. If all the data are mapped to one processor, 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 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 examples below illustrate simple cases of communication, parallelism, and their interaction. Note that the examples used are chosen for illustration and do not necessarily reflect efficient data layouts or computational methods for the program fragments shown.

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 and does not necessarily reflect the actual compilation process.

Consider the statements below:

```
      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).

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.

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.

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. 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 the last, 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 these two pairs of references is unknown.

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 may contribute to the total inherent 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 statement 4 is executed on the processor storing `c(i)`. (This is an optimal strategy for this example, although not for others.) Then:

- 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, although it is easy for a compiler to notice the update and remember the value; and
- Element `c(i+3)` *does not* induce new communication, since it was used in statement 1 and not changed since.

Thus, the total inherent communication in this program fragment is four array elements.

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 this does 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 statements below:

```
      REAL a(1000), b(1000), c(1000)
!HPF$ PROCESSORS procs(10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: a, b, 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. 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 document.

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)` (where $\ominus$ and $\oplus$ represent base-10 "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, 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.

Some 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 a(BLOCK), d(BLOCK,*) ONTO procs
      ...
      a(1:200) = a(1:200) + a(2:400:2)
      c = c + d
```

In this example, the use of different strides in the two references to a will cause communication to align the two sections on the right-hand side of the first statement. The second assignment statement requires a latent transpose.

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 wil also be mo ed to new locations on the same processor. The communication vo me is th same as rray assignment from an array with the original alignment to another array with the ne lignment. The REDISTRIBUTE statement changes the distribution for every array aligned to a template. Therefore, its cost is similar to the cost of a REALIGN on many arrays simultaneously. Advanced 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(1000), y(1000)
!HPF$ DISTRIBUTE x(BLOCK), y(BLOCK) ONTO P
      ...
      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$ DISTRIBUTE x(BLOCK), y(BLOCK), z(BLOCK) ONTO P
      ...
!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.[1] Therefore, the loop has substantial potential parallelism and will likely execute much faster than the last example.

Partitioning a computation may itself require communication. Consider the following code:

[1]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.

```
      INTEGER indx(1000), inv(1000)
!HPF$ DISTRIBUTE indx(BLOCK), inv(BLOCK) ONTO P

      FORALL ( j = 1:1000 ) inv(indx(j)) = j**2
```

Since the location of the reference 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 squares for all elements for which it received notification.

The optimal sharing scheme, its implementation and its cost will be highly architecture-dependent.

A specified data distribution could create a trade-off between exploitable parallelism and communication overhead. Consider the following code:

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

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 a(*,BLOCK), b(*,BLOCK)
  ```

  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.

- Divide each column 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, the operation can be implemented 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$ DISTRIBUTE (CYCLIC) :: 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

A goal of the HPF design is that HPF directives be 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 could be to remove the comment character and directive prefix from each directive.

hpf-directive-line	**is**	*directive-origin hpf-directive*
directive-origin	**is**	**!HPF$**
	or	**CHPF$**
	or	***HPF$**
hpf-directive	**is**	*declarative-directive*
	or	*executable-directive*
declarative-directive	**is**	*processors-directive*
	or	*view-directive*
	or	*align-directive*
	or	*distribute-directive*
	or	*dynamic-directive*
	or	*inherit-directive*
	or	*template-directive*
	or	*combined-directive*
	or	*pure-directive*
	or	*extrinsic-directive*
	or	*local-directive*
	or	*sequence-directive*
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 *declarative-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)

Constraint: An *hpf-directive* 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* may only be the characters **!HPF$** in free source form. 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.

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 (note that column 6 must be blank, except when signifying continuation) is:

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

An example of 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) is:

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

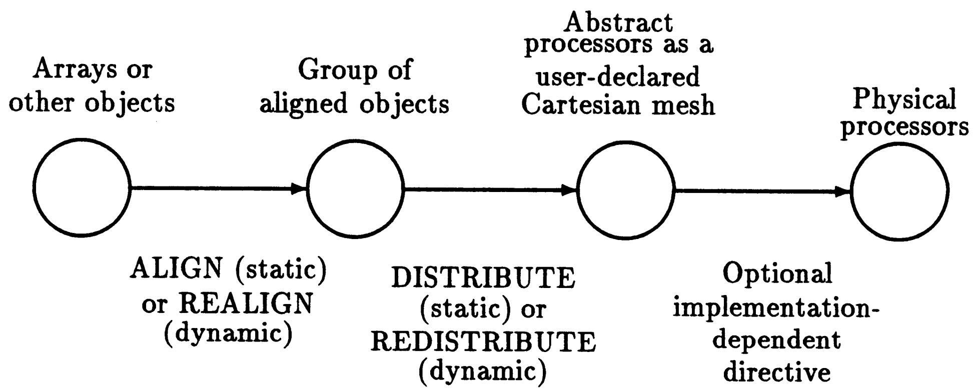
Chapter 3

Data Alignment and Distribution Directives

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 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 language-processor-dependent.)

The following diagram illustrates the model:


Arrays or other objects
Group of aligned objects
Abstract processors as a user-declared Cartesian mesh
Physical processors
ALIGN (static) or REALIGN (dynamic)
DISTRIBUTE (static) or REDISTRIBUTE (dynamic)
Optional implementation-dependent directive

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, pretty much 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 general idea is that every array (indeed, every object) is created with *some* distribution onto *some* arrangement of 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 be may be specified explicitly. In either case, any such explicit declarative information is used when the array is created. (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.)

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.

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.

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 language-processor-dependent, but must be expressible as explicit directives for that implementation. 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 below.)

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 is 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.

Alignment is considered an *attribute* (in the Fortran 90 sense) of an array or scalar. Distribution is technically an attribute of the index space of the array. Sometimes we speak loosely of the distribution of an array, but this really means the distribution of the index space of the array, or of another array to which it is aligned. The relationship of an array to a processor arrangement is properly called the *mapping* of the array. (Even more technically, 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.)

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 provides the notion of 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 attributed form analogous to type declaration statements in Fortran 90 using the "::" punctuation.

The attributed 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 any of them, be a type specifier.

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

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

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

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

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

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.

Fortran 90 attributes appearing in an HPF directive must also be declared by standard Fortran 90 statements (for the practical reason that Fortran processors that ignore HPF directives would not observe the attribute information).

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 processor, or a replicated mapping, wherein each element of an array may be mapped to many 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 3.11).
- 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 declaratively 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 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 processors. (There must be at least $\lceil 10000/256 \rceil = 40$ 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 processors, then 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 processors in round-robin fashion.

Distributions may be specified independently for each axis 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 processors. The GO_BOARD array will have its rows distributed cyclically over a one-dimensional arrangement of 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.)

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 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 effort at run time; the programmer must take care when using this feature.)

The DISTRIBUTE directive may appear only in the *declaration-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:

distribute-directive	**is**	DISTRIBUTE *distributee dist-attribute-stuff*
redistribute-directive	**is**	REDISTRIBUTE *distributee dist-attribute-stuff*
	or	REDISTRIBUTE *dist-attribute-stuff* :: *distributee-list*
dist-attribute-stuff	**is**	*dist-format-clause* [*dist-onto-clause*]
	or	*dist-onto-clause*
distributee	**is**	*object-name*
	or	*template-name*

dist-format-clause	is	(*dist-format-list*)
	or	* (*dist-format-list*)
	or	*
dist-format	is	`BLOCK` [(*int-expr*)]
	or	`CYCLIC` [(*int-expr*)]
	or	*
dist-onto-clause	is	`ONTO` *dist-target*
dist-target	is	*processors-name*
	or	* *processors-name*
	or	*

Constraint: If either the *dist-format-clause* or the *dist-target* begins with *, then the directive must be `DISTRIBUTE` (rather than `REDISTRIBUTE`) and every *distributee* must be a subprogram dummy argument.

Constraint: An *object-name* mentioned as a *distributee* may not appear as an *alignee* in an `ALIGN` or `REALIGN` directive.

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 processors 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 processors 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: 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 are split across more than one abstract processor.

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

Examples:

```
!HPF$ DISTRIBUTE D1(BLOCK)
!HPF$ DISTRIBUTE (BLOCK,*,BLOCK) ONTO P:: 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`.

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

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 indexing on all arrays is 1-based. Then `BLOCK(`m`)` means that a distributee position whose index along that dimension is j is mapped to a 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 processor. The first distributee position in processor k along that axis is position number `1+`m`*(`k`-1)`.

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

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

`CYCLIC(`m`)` and `BLOCK(`m`)` imply the same distribution when $m \cdot p \geq d$, but `BLOCK(`m`)` additionally asserts that the distribution will not wrap around in a cyclic manner, which a compiler might not be able to determine at compile time if m is an expression. 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.

The statement form of a `DISTRIBUTE` or `REDISTRIBUTE` directive may be considered a mellisonant abbreviation for an attributed form that happens to mention only one alignee; 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 alignment of scalars.

If the *dist-format-clause* is omitted from the attributed form, it is assumed to consist of a parenthesized list of "`BLOCK`" entries, equal in number to the rank of the distributees. So the directive

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

means

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

which in turn means the same as

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

In either the statement form or the attributed form, if the `ONTO` clause is present, it specifies the processor array that is the target of the distribution. If the `ONTO` clause is omitted, then a language-processor-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 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 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 effort at run time; the programmer must take care when using this feature.)

The `ALIGN` directive may appear only in the *declaration-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 if `ALIGN` and `REALIGN` is as follows:

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

align-attribute-stuff	**is**	[(*align-source-list*)] *align-with-clause*
alignee	**is**	*object-name*
align-source	**is**	:
	or	*
	or	*align-dummy*
align-dummy	**is**	*scalar-int-variable*

Constraint: An *object-name* mentioned as an *alignee* may not appear as a *distributee* in a DISTRIBUTE or REDISTRIBUTE directive.

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

Constraint: The *align-source-list* (and its surrounding parentheses) must be omitted if the alignee is scalar. (In some cases this will preclude the use of the statement form of the directive.)

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

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.)

The statement form of an ALIGN or REALIGN directive may be considered a mellivident 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:

align-with-clause	**is**	`WITH` *align-spec*
align-spec	**is**	*align-target* [(*align-subscript-list*)]
	or	* *align-target* [(*align-subscript-list*)]
align-target	**is**	*object-name*
	or	*template-name*
align-subscript	**is**	*int-expr*
	or	*align-subscript-use*
	or	*subscript-triplet*
	or	*
align-subscript-use	**is**	[[*align-subscript-use*] *add-op*] *align-add-operand*
align-add-operand	**is**	[*align-add-operand* *] *align-mult-operand*
align-mult-operand	**is**	*align-dummy*
	or	(*align-subscript-use*)
	or	*int-mult-operand*
align-subscript-use-subset	**is**	[*add-op*] *align-product* [*add-op* *int-add-operand*]
	or	[*add-op*] *int-mult-operand* *add-op* *align-product*
align-product	**is**	[*int-add-operand* *] *align-dummy*

Constraint: If the *align-spec* begins with "`*`", then the directive must be `ALIGN` (rather than `REALIGN`) and every *alignee* must be a subprogram dummy argument.

Constraint: The *int-mult-operand* used as the last alternative for an *align-mult-operand* must not contain any occurrences of an *align-dummy*.

Constraint: An *align-subscript-use* expression may contain occurrences of at most one *align-dummy* (but there may be multiple occurrences of that one dummy name).

Constraint: An *align-dummy* variable may not appear anywhere in the *align-spec* except where explicitly permitted to appear by virtue of the grammar shown above.

Constraint: Implementations of the HPF subset may require that *align-subscript-use* expressions obey the more restrictive syntax described by the rule for *align-subscript-use-subset.*

The syntax rules for an *align-subscript-use* are a bit tricky because of operator precedence issues, but the basic idea is simple: an *align-subscript-use* is intended to be a linear function of a single *align-dummy* variable. In the subset, the expression must have the explicit form of a single occurrence of an *align-dummy*, possibly negated or multiplied by an integer expression and then possibly added to (or subtracted from) an integer expression. In the full HPF language, the single *align-dummy* variable may appear more than once in an *align-subscript-use* expression, but the expression is syntactically limited to forms that can be reduced to an explicitly linear form through appropriate algebraic transformations without cancellation.

For example, the following *align-subscript-use* expressions are valid in the subset (assuming that J, K, and M are *align-dummy* variables 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)
```

The following *align-subscript-use* expressions are valid in the full HPF language but not in the subset:

```
J+J  2*(J+1)     3*K-2*K  M*2    M*N      10000-M*3
J-J  2*J-3*J+J   5-K+3    M+2-3  N*(M-N)  2*(3*(K-1)+13)-100
```

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

```
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)
```

Note that even though 2**(2*J-3*J+J) happens to represent a (degenerate) linear function of J, it is regarded as syntactically invalid for use as an *align-subscript-use* expression.

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

$$(UA\text{-}LA\text{+}1)\ \texttt{.EQ.}\ \texttt{MAX}((UT\text{-}LT\text{+}ST)/ST,0)$$

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.*

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(:)`

Please 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 to hand. (Of course, when a replicated data object is written, all copies must be updated, not just one copy. Replicated representations are very useful for such tricks as small lookup tables, where it much faster to have a copy in each physical processor but you don't want to be bothered giving it an extra dimension that is logically unnecessary to the algorithm.)

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*. The selected element of the *alignee* is then considered to be aligned with the indicated element of the *align-target*; or, more accurately, 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 multiple-align, the alignees must all have the same rank but need not all have the same shape; the sizes 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
```

The idea here is that you know there are processors (perhaps N of them) and you want arrays of different sizes (3, 4, 43) within each processor. It's okay as far as HPF is concerned for the numbers 3, 4, and 43 to 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
!HPF$ ALIGN X WITH D2
```

3.5 DYNAMIC Directive

dynamic-directive	**is**	DYNAMIC *alignee-or-distributee-list*
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. (If you want to get this kind of effect, use Fortran 90 modules 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, more or less consistent with 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 other 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(:) :: 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.

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
      POINTER,REAL :: 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 easy for a compiler to notice that it will immediately be remapped to a `CYCLIC` distribution. Similar remarks apply to `TINKER` and `EVERS`.

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.

(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.)

When an array is allocated, it will be aligned to an existing index space 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 size of each dimension.

It may only appear in the *declaration-part* of a scoping unit. The product of the dimensions must be greater than zero.

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.

If two processor arrangements have the same shape and neither has the `VIEW` attribute, then corresponding elements of the two arrangements are understood to refer to the same abstract processor. The use of the `VIEW` attribute may also cause an element of one processor arrangement to refer to the same abstract processor as an element of some other arrangement. (It is anticipated that language-processor-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 point during the program execution, the intent is that the two objects be mapped to the same physical processor at that point.

The intrinsics `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 processors arrangements.

processors-directive	**is**	`PROCESSORS` *processors-decl-list*
processors-decl	**is**	*processors-name* [(*explicit-shape-spec-list*)]
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 processors arrangement is conceptually scalar.

(Rationale: A scalar processors 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 processors 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 processors arrangement might be mapped to some arbitrarily chosen computational processor or replicated onto all computational processors.)

An HPF compiler is required to accept any PROCESSORS declaration in which the product of the dimensions of each declared processors 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 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 forbidden 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, identical upper bounds, and identical view attribute information (if any).

  (Note that 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.)

Variables in COMMON or having the SAVE attribute may be mapped to a locally declared processors 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 processors arrangements to which they may be mapped to depend on the value returned by NUMBER_OF_PROCESSORS.

The remainder of this section is advice to implementors and is not part of HPF:

> 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 an language-processor-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 an language-processor-dependent directive or through the programming environment (for example, as a UNIX command-line argument); again, 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.

3.8 VIEW Directive

The VIEW attribute provides a mechanism to allow the same set of abstract processors to be viewed as having different rectilinear geometries, possibly of differing rank. (This feature is sometimes loosely called "EQUIVALENCE for processors arrangements.")

view-directive	is	VIEW *processor-view view-attribute-stuff*
view-attribute-stuff	is	OF *processor-viewed*
processor-view	is	*processor-name* [*permutation*]
processor-view-entity	is	*processor-name* [(*array-spec*)] [*permutation*]
processor-viewed	is	*processor-name* [*permutation*]
permutation	is	*array-constructor*

Constraint: A permutation, if present, must be a constant integer array, syntactically expressed as an array constructor, whose elements are a permutation of the values

$(1, 2, \ldots, n)$, where n is the rank of the associated processor-name. If it is omitted and the processor-name names a non-scalar processors arrangement, it is as if the identity permutation `(/ 1, 2, ..., n /)` had been specified.

Constraint: The `VIEW` directive may only appear in a *declaration-part* of a program.

The `VIEW` directive relates each of a list of processors arrangement names, each with an optional permutation array, to one specific other processors arrangement, which may also have a permutation array. The relationship between a view A and a viewed arrangement B is established as follows. If A is scalar, B must be scalar, and the permutation must be absent or empty; in this case A and B designate the same abstract processor. If A is non-scalar, B must be non-scalar, with the same size as A but not necessarily the same shape or rank. Let P be the permutation array for A and Q be the permutation array for B; let M be the rank of A (and thus the length of P) and let N be the rank of B (and thus the length of Q). Permute the dimensions of A, yielding a processor array A' such that dimension J of A' corresponds to dimension $P(J)$ of A for $1 \leq J \leq M$. Similarly permute the dimensions of B, yielding a processor array B' such that dimension K of B' corresponds to dimension $Q(K)$ of B for $1 \leq K \leq N$. The permuted processor arrays A' and B' are then "equivalenced" using Fortran's usual column-major order.

```
      PARAMETER (A = (/2,1/))

!HPF$ PROCESSORS P(10,10), Q(100), R(100)
!HPF$ PROCESSORS S(10,10), T(100)

!HPF$ VIEW OF P :: Q
!HPF$ VIEW OF P(/2,1/) :: R

!HPF$ VIEW S(/A/) OF T
```

In the code fragment above, the processor arrays `P`, `Q` and `R` designate the same set of 100 abstract processors in different ways. Because `P` has no `VIEW` attribute, it designates the same two-dimensional arrangement as any other 10×10 processor arrangement with no `VIEW` attribute.

The first `VIEW` directive specifies that `Q` names the same set of processors as `P`, in such a way that, for all `I` in the range `1:10` and all `J` in the range `1:10`, `P(I,J)` and `Q((J-1)*10+I)` designate the same processor.

The second `VIEW` directive specifies that `R` names the same set of processors as `P` after permuting the dimensions of the latter. Thus, in this case `P(I,J)` and `R((I-1)*10+J)` designate the same processor.

The third `VIEW` directive specifies that `S`, taken in row-major order, provides a view of the one-dimensional processor arrangement `T`. A named parameter constant `A` is used to specify the permutation.

The `VIEW` attribute may appear in a combined-directive such as

```
!HPF$ PROCESSORS WILL_YOU_STILL_NEED_ME__WHEN_IM(64)
!HPF$ PROCESSORS, VIEW OF WILL_YOU_STILL_NEED_ME__WHEN_IM,    &
!HPF$       DIMENSION(4,4,4) :: RUBIKS_REVENGE
!HPF$ PROCESSORS,VIEW OF RUBIKS_REVENGE(/2,3,1/) ::       &
!HPF$             CHESSBOARD(8,8)(/2,1/)
```

For all I in the range 1:8 and all J in the range 1:8, CHESSBOARD(I,J) means the same processor as RUBIKS_REVENGE((I-1)/2+1, IAND(J-1,3)+1, 2*IAND(I-1,1)+(J-1)/4+1).

3.9 INHERIT Directive

inherit-directive	is	INHERIT *dummy-argument-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 may not have both the INHERIT attribute and the ALIGN attribute. The INHERIT directive in a may only appear in a *declaration-part* of a program.

The INHERIT attribute specifies that the index space (template) for a dummy argument should be inherited, by making it a copy of the index space of the actual argument. (The actual argument must be a whole array or an array section; it may not be an expression of any other form.) Moreover, the INHERIT attribute implies a default distribution of DISTRIBUTE * ONTO *. See section 3.11 for further exposition.

Without the INHERIT attribute, the index space of a dummy argument has the same shape as the dummy itself and the dummy argument is aligned to its index space 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.

3.10 TEMPLATE Directive

The TEMPLATE directive declares one or more templates, specifying for each the name, the rank (number of dimensions), and the size of each dimension. It must appear in the declaration-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 long list in section 12.1.2.2.1 of the Fortran 90 standard.

A template is simply an abstract space of indexed positions; you can think of it as an "array of nothings" (as compared to an "array of integers," say). If an array is a cat, then a template is a Cheshire cat, and the index space is the grin. A template may be used as an abstract *align-target* that may then be distributed.

template-directive	is	TEMPLATE *template-decl-list*
template-decl	is	*template-name* [(*explicit-shape-spec-list*)]
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 odd 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 an index space 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 3.11).

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 (index space) 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 actual argument must be 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 forbidden 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).

  (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.)

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.

3.11 Alignment, Distribution, and Subprogram Interfaces

Alignment directives may be applied to dummy arguments in the same manner as for other variables. 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.

Suppose that we wish to speak explicitly about the distribution of a dummy argument. First we must decide what is the index space that is subject to distribution. A dummy argument always has a fresh index space (a template) to which it is ultimately aligned; this index space is constructed in one of two ways. If the dummy argument 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 has the INHERIT attribute, then the index space is "inherited" from the actual argument according to the following rules:

- If there is an explicit interface for the called subprogram and that interface contains prescriptive directives for the dummy argyment in question, the actual argument will be remapped if necessary to conform to the directives in the explicit interface. The index space of the dummy will then be as declared in the interface.
- Otherwise:
  - If the actual argument is a whole array, the index space of the dummy is a copy of the index space with which the actual argument is ultimately aligned.
  - If the actual argument is an array section, let A be the array; the index space of the dummy is a copy of the index space with which A is ultimately aligned.
  - If the actual argument is any other expression, the index space may be chosen arbitrarily by the language processor.

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(1024),TWIST(1024)
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ DISTRIBUTE (BLOCK) ONTO DANCE_FLOOR::FRUG,TWIST
      CALL TERPSICHORE(FRUG(1:400:3),TWIST(1:400:3))
```

The two array sections FRUG(1:400:3) and TWIST(1:400:3) are mapped onto processors in the same manner. 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
```

For now, suffice it to say that within subroutine TERPSICHORE it would be correct to declare

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

but it would not be correct to declare

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

Each of these asserts that the index space of the specified dummy argument is already distributed BLOCK on entry to the subroutine. The index space for TANGO is 1:1024, inherited (copied) from the array TWIST, whose section was passed as the corresponding actual argument, and that index space does indeed have a BLOCK distribution. But the index space for FOXTROT is 1:134; the layout of the elements of the actual argument FRUG(1:400:3) (22 on the first processor, 21 on the second processor, 21 on the third processor, 22 on the fourth processor, ...) cannot properly be described as a BLOCK distribution of a length-134 index space, so the DISTRIBUTE declaration for FOXTROT shown above would indeed be erroneous. (On the other hand, the layout of FRUG(1:400:3) can be described in terms of an alignment to a length-1024 index space, which can be described by declaring a template (see section 3.10), so the directives

```
!HPF$ PROCESSORS DANCE_FLOOR(16)
!HPF$ TEMPLATE,DISTRIBUTE(BLOCK) ONTO DANCE_FLOOR::GURF(1024)
!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.)

The simplest case is the use of the INHERIT attribute alone. If a dummy argument has the INHERIT attribute and no explicit ALIGN or DISTRIBUTE attribute, the net effect is to tell the compiler to leave the data exactly where it is—don't 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 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.)

Let us now consider 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 accurately 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.)

- 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.)

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.

Consider, then, these amusing examples of DISTRIBUTE directives for dummies:

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

Compiler, please do whatever it takes to cause URANIA to have a CYCLIC distribution on the processors arrangement GALILEO.

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

Compiler, please do whatever it takes to cause POLYHYMNIA to be distributed onto the processors arrangement ELVIS, using whatever distribution format it currently has (which might be on some other processors arrangement).

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

Compiler, please do whatever it takes to cause THALIA to have a CYCLIC distribution on the processors arrangement FLIP; for whatever it's worth, I claim that THALIA already has a cyclic distribution, though it might be on some other processors arrangement.

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

Compiler, please do whatever it takes to cause CALLIOPE to have a CYCLIC distribution on the processors arrangement HOMER; for whatever it's worth, I claim that CALLIOPE is already distributed onto HOMER, though it might be with some other distribution format.

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

Compiler, I claim that MELPOMENE is already distributed onto EURIPIDES; use whatever distribution format the actual had (and I hope that means you won't have to move any data around).

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

Compiler, I claim that CLIO is already distributed CYCLIC onto HERODOTUS and I like it that way (so I hope you won't have to move any data around).

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

Compiler, please do whatever it takes to cause EUTERPE to have a CYCLIC distribution onto whatever processors arrangement the actual was distributed onto.

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

Compiler, just leave ERATO alone. Please. Wherever the actual argument was, that's a good place for ERATO.

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

Compiler, I claim that ARTHUR_MURRAY is already distributed CYCLIC onto whatever processors arrangement the actual was distributed onto. I like it that way (so I hope you won't have to move any data around).

One may also omit either the *dist-format-clause* or the *dist-target-clause* for a dummy. If such a clause is omitted and the dummy 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 processors arrangement arbitrarily. Examples:

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

Compiler, I claim that DAVID_LETTERMAN is already distributed on TV, though I'm not saying how. As far as I'm concerned, you can change the distribution format as long as you keep him on TV.

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

Compiler, I claim that WHEEL_OF_FORTUNE is already CYCLIC. As long as you keep it CYCLIC, you're free to remap it onto some other processors arrangement (though I can't imagine why you'd want to).

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

If the dummy 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, then the programmer cannot assert a claim about the distribution of the natural template of a dummy with any certitude of portability.

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

It is perfectly clear that 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, while the second dummy argument, CZAR, is forced to be aligned with the first dummy. If the two actuals are already aligned, no remapping of the data will be required at run time; but the subprogram will operate correctly even if the actuals are not already aligned, at the cost of remapping the data for the second dummy argument at run time.

If the *align-spec* is "*" or 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$ 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.

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".

Up to this point we have spoken about dummy arguments as if the REALIGN and REDISTRIBUTE directives did not exist. Here are the rules on the interaction of these directives with the subprogram argument interface.

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 index space.

2. If an array or any section thereof is accessible by two or more paths, it is illegal 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            !Illegal
      ...
      END
```

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

Chapter 4

Statements

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 array assignments and WHERE constructs 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, FORALL may call user-defined functions on the elements of an array, which is not possible in Fortran 90 array assignments. Both single-statement and block FORALL forms are defined in this chapter.

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 standard 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 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)
```

Later examples will show other uses of FORALL which are difficult to express using array assignment or other means.

The FORALL 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; however, more general array sections can be assigned in FORALL, and functions can be called within the expression. It is important to note that FORALL is not intended to be a general parallel construct; for example, it does not express functional parallelism or pipelined computations well. This was an explicit design decision made in order to simplify the construct and promote agreement on the statement's semantics.

4.1.1 General Form of Element Array Assignment

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

forall-stmt	**is**	FORALL *forall-header forall-assignment*
forall-header	**is**	(*forall-triplet-spec-list* [, *scalar-mask-expr*])
forall-triplet-spec	**is**	*index-name* = *subscript* : *subscript* [: *stride*]
forall-assignment	**is**	*assignment-stmt*
	or	*pointer-assignment-stmt*

Constraint: Any procedure referenced in a *forall-stmt*, including one by a defined operation or assignment in the *forall-assignment*, must be a pure function, as defined in Section 4.3, and is syntactically guaranteed not to have side effects.

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

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

For each *index-name* in the *forall-assignment*, the set of permitted values is determined on entry to the statement and is $m1 + (k-1) \times m3$, $k = 1, 2, ..., \left\lfloor \frac{m2-m1+m3}{m3} \right\rfloor$ and where *m1*, *m2*, and *m3* are the values of the first subscript, the second subscript, and the stride respectively in the *forall-triplet-spec*. If *stride* is missing, it is as if it were present with a value of the integer 1. The expression *stride* must not have the value 0. If for some *index-name* $\lfloor (m2 - m1 + m3)/m3 \rfloor \leq 0$, the *forall-assignment* is not executed.

Examples of the FORALL statement syntax are:

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

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

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

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* or *target* and all subscripts contained in the *array-element* or *array-section* in 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 elements specified by *array-element* or *array-section*. The assignments 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 *array-element* 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.

The *forall-stmt* must not cause any atomic data object to be assigned a value more than once. A data object is atomic if it contains no subobjects; thus, an integer variable is an atomic object, but an array of integers is an object that is not atomic. An assignment to a non-atomic object is considered to also assign to all subobjects of that object. A pointer assignment is considered to assign to the pointer object, not to the target of the pointer. The *forall-stmt* assigns to all data objects assigned by its *forall-assignment.*

Since a function called from a FORALL construct 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 when all functions are declared pure.

4.1.3 Examples of FORALL Statement Interpretation

The FORALL statements

```
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)
```

each 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))
```

The FORALL statement

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

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.

The FORALL statement

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

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.

The FORALL statement

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

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

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.

If the FORALL statement

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

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
```

The FORALL statement

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

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 is also used.

The FORALL statement

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

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, 5 /)
```

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

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

after its execution.

The FORALL statement

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

computes nine sums of subarrays of x. (SUM is allowed in a FORALL because it is an intrinsic function, and 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. 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 an 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.

A *forall-stmt* of the general form

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

is equivalent to the following code similar to Fortran 90:

```
! Evaluate subscript and stride expressions.
! These assignments may be executed in any order.
$templ_1$ = $l_1$
$tempu_1$ = $u_1$
$temps_1$ = $s_1$
$templ_2$ = $l_2$
$tempu_2$ = $u_2$
$temps_2$ = $s_2$
  ...
$templ_n$ = $l_n$
$tempu_n$ = $u_n$
$temps_n$ = $s_n$

! 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.
DO $v_1$=$templ_1$,$tempu_1$,$temps_1$
  DO $v_2$=$templ_2$,$tempu_2$,$temps_2$
    ...
    DO $v_n$=$templ_n$,$tempu_n$,$temps_n$
      tempmask($v_1$,$v_2$,...,$v_n$) = mask
      IF (tempmask($v_1$,$v_2$,...,$v_n$)) THEN
        temprhs($v_1$,$v_2$,...,$v_n$) = rhs
        $tempe_1$($v_1$,$v_2$,...,$v_n$) = $e_1$
        $tempe_2$($v_1$,$v_2$,...,$v_n$) = $e_2$
        ....
        $tempe_m$($v_1$,$v_2$,...,$v_n$) = $e_m$
      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.
DO $v_1$=$templ_1$,$tempu_1$,$temps_1$
  DO $v_2$=$templ_2$,$tempu_2$,$temps_2$
    ...
    DO $v_n$=$templ_n$,$tempu_n$,$temps_n$
      IF (tempmask($v_1$,$v_2$,...,$v_n$)) THEN
        a($tempe_1$($v_1$,$v_2$,...,$v_n$),...,$tempe_m$($v_1$,$v_2$,...,$v_n$)) =      &
```

```
                temprhs(v1, v2, ..., vn)
            END IF
        END DO
      ...
    END DO
  END DO
```

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 the assignments to both arrays *temprhs* and a must be changed to pointer assignments (and 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.

4.1.5 Consequences of the Definition of the FORALL Statement

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, we have not stated it as a syntax constraint.

Right-hand sides and subscripts 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* is 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.

4.2 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.*

executable-construct	**is**	. . .
	or	*forall-construct*
forall-construct	**is**	**FORALL** *forall-header*
		forall-body-stmt-list
		END FORALL
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-construct*, including one referenced by a defined operation or assignment in a *forall-body-stmt*, must be a pure function as defined in Section 4.3.

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

For each *index-name* in the *forall-assignments*, the set of permitted values is determined on entry to the construct and is

$$m1 + (k-1) * m3, where\ k = 1, 2, ..., \left\lfloor \frac{m2 - m1 + m3}{m3} \right\rfloor$$

and where *m1*, *m2*, and *m3* are the values of the first subscript, the second subscript, and the stride respectively in the *forall-triplet-spec.* If *stride* is missing, it is as if it were present with a value of the integer 1. The expression *stride* must not have the value 0. If for some *index-name* $\lfloor (m2 - m1 + m3)/m3 \rfloor \leq 0$, the *forall-assignments* are not executed.

Examples of the **FORALL** construct are:

```
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

FORALL ( i=1:n-1 )
  FORALL ( j=i+1:n )
    a(i,j) = a(j,i)      ! make a be symmetric
  END FORALL
END FORALL

FORALL ( i=1:n, j=1:n )
  a(i,j) = MERGE( a(i,j), a(i,j)**2, i.EQ.j )
  WHERE ( .NOT. done(i,j,1:m) )
    b(i,j,1:m) = b(i,j,1:m)*x
  END WHERE
END FORALL
```

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 right-hand side *expr* and any left-hand side subscripts for all active *index-name* values, then assign the right-hand side results to the corresponding left-hand side references.

   (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.

Each *forall-assignment* must obey the same restrictions in a *forall-construct* as in a simple *forall-stmt*. (Note that any innermost statement within nested FORALL constructs must always be a *forall-assignment*.) In addition, each *where-stmt* or assignment nested within a *where-construct* must obey these restrictions. 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 Scalarization of the FORALL Construct

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.

A *forall-construct* of the form:

```
FORALL (... $e_1$ ... $e_2$ ... $e_n$ ...)
    $s_1$
    $s_2$
     ...
    $s_n$
END FORALL
```

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

```
$temp_1$ = $e_1$
$temp_2$ = $e_2$
 ...
$temp_n$ = $e_n$
FORALL (... $temp_1$ ... $temp_2$ ... $temp_n$ ...) $s_1$
FORALL (... $temp_1$ ... $temp_2$ ... $temp_n$ ...) $s_2$
   ...
FORALL (... $temp_1$ ... $temp_2$ ... $temp_n$ ...) $s_n$
```

A similar remark can be made when the s_i may be **WHERE** or **FORALL** constructs.

A *forall-construct* of the form:

```
FORALL ( $v_1$=$l_1$:$u_1$:$s_1$, $mask_1$ )
  WHERE ( $mask_2$ )
    a($l_2$:$u_2$:$s_2$) = $rhs_1$
  ELSEWHERE
    a($l_3$:$u_3$:$s_3$) = $rhs_2$
  END WHERE
END FORALL
```

is equivalent to the following code:

```
! Evaluate subscript and stride expressions.
! These assignments can be made in any order.
$templ_1$ = $l_1$
$tempu_1$ = $u_1$
$temps_1$ = $s_1$

! Evaluate the FORALL mask expression.
! 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 bounds and masks for the WHERE.
! The iterations of this loop may be executed in any order.
! The assignments in the loop body may be executed in any order,
! provided the mask bounds and stride are computed before the mask.
! The loop body need not be executed atomically.
DO $v_1$=$templ_1$,$tempu_1$,$temps_1$
  IF ($tempmask$($v_1$)) THEN
    $tempmask_2$($v1$) = $mask_2$
  END IF
END DO

! Evaluate the WHERE branch.
! The iterations of this loop may be executed in any order.
DO $v_1$=$templ_1$,$tempu_1$,$temps_1$
  IF ($tempmask$($v_1$)) THEN
    $tmpl_2$($v_1$) = $l_2$
    $tmpu_2$($v_1$) = $u_2$
    $tmps_2$($v_1$) = $s_2$
    WHERE ( $tempmask_2$ )
      $temprhs_1$($tmpl_2$($v_1$):$tmpu_2$($v_1$):$tmps_2$($v_1$)) = $rhs_1$
    END WHERE
  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 ($tempmask$($v_1$)) THEN
    WHERE ( $tempmask_2$ )
      $a$($tmpl_2$($v_1$):$tmpu_2$($v_1$):$tmps_2$($v_1$)) = &
        $temprhs_1$($tmpl_2$($v_1$):$tmpu_2$($v_1$):$tmps_2$($v_1$))
    END WHERE
  END IF
END DO

! Evaluate the ELSEWHERE branch.
! The iterations of this loop may be executed in any order.
DO $v_1$=$templ_1$,$tempu_1$,$temps_1$
  IF ($tempmask$($v_1$)) THEN
    $tmpl_3$($v_1$) = $l_3$
    $tmpu_3$($v_1$) = $u_3$
    $tmps_3$($v_1$) = $s_3$
    WHERE ( .NOT. $tempmask_2$ )
      $temprhs_2$($tmpl_3$($v_1$):$tmpu_3$($v_1$):$tmps_3$($v_1$)) = $rhs_2$
    END WHERE
    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 ($tempmask$($v_1$)) THEN
      WHERE ( .NOT. $tempmask_2$ )
        $a$($tmpl_3$($v_1$):$tmpu_3$($v_1$):$tmps_3$($v_1$)) = &
          $temprhs_2$($tmpl_3$($v_1$):$tmpu_3$($v_1$):$tmps_3$($v_1$))
      END WHERE
    END IF
  END DO
```

Note that the assignments to *tempmask*$_2$ are array assignments and require special treatment (including saving of bounds and stride 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 will generate two loops similar to those shown above.

A *forall-construct* of the form:

```
  FORALL ( $v_1$=$l_1$:$u_1$:$s_1$, $mask_1$ )
    FORALL ( $v_2$=$l_2$:$u_2$:$s_2$, $mask_2$ )
      $a$($e_1$) = $rhs_1$
      $b$($e_2$) = $rhs_2$
    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 ($tempmask$($v_1$)) THEN
      DO $v_2$ = $templ_2$($v_1$),$tempu_2$($v_1$),$temps_2$($v_1$)
        IF ( $tempmask_2$($v_1$,$v_2$) ) THEN
          $a$($tmpe_1$($v_1$,$v_2$)) = $temprhs_1$($v_1$,$v_2$)
        END IF
      END DO
    END IF
  END DO

  ! Evaluate second statement.
  ! Ordering constraints are as for the first statement.
  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
          $temprhs_2$($v_1$,$v_2$) = $rhs_2$
          $tmpe_2$($v_1$,$v_2$) = $e_2$
        END IF
      END DO
    END IF
  END DO
  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
          $b$($tmpe_2$($v_1$,$v_2$)) = $temprhs_2$($v_1$,$v_2$)
      END IF
    END DO
  END IF
END DO
```

Again, the extensions to higher dimensions are straightforward, as is the extension to deeper nesting levels. Each statement at the deepest nesting level will generate two loops of the types shown.

4.2.4 Examples of FORALL Construct Interpretation

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
```

is equivalent to the two Fortran 90 statements

```
a(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) = a(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.

The **FORALL** construct

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

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.

The **FORALL** construct

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

when executed with the input arrays

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

will produce as results

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

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

4.2.5 Consequences of the Definition of the FORALL Construct

A block FORALL means roughly the same as 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.

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 I/O. A *pure subroutine* is one that produces no side effects except for modifying the values and/or pointer associations of certain arguments.

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:

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

The freedom from side effects of a pure function ensures that it can be invoked concurrently in a `FORALL` without such undesirable consequences as nondeterminism, and additionally assists the efficient implementation of concurrent execution.

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 both its interface body (if provided) and its definition must contain the `PURE` declaration. The form of this declaration is a directive immediately after the *function-stmt* or *subroutine-stmt* of the procedure interface body or definition:

pure-directive **is** `!HPF$ PURE` [*procedure-name*]

Intrinsic functions, including HPF intrinsic functions, are always pure and require no explicit declaration of this fact; intrinsic subroutines are pure if they are elemental (e.g., `MVBITS`) but not otherwise. A statement function is pure if and only if all functions that it references are pure.

Pure function definition

To define pure functions, Rule R1215 of the Fortran 90 standard is changed to:

function-subprogram **is** *function-stmt*
[*pure-directive*]
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

with the following constraints in addition to those in Section 12.5.2.2 of the Fortran 90 standard:

Constraint: If a *procedure-name* is present in the *pure-directive*, it must match the *function-name* in the *function-stmt*.

Constraint: In a pure function, a local variable must not have the `SAVE` attribute. (Note that this means that a local variable cannot be initialised in a *type-declaration-stmt* or a *data-stmt*, which imply the `SAVE` attribute.)

Constraint: A pure function must 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 a *index-name* in a *forall-triplet-spec*;
- 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 `(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: In a pure function, a dummy argument must not appear in an explicit mapping directive unless it has the `INHERIT` attribute.

Constraint: In a pure function, 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 function, a dummy argument or local 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 *pause-stmt*, *stop-stmt*, or I/O statement (including a file operation).

To declare that a function is pure, a *pure-directive* must be given.

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 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 global objects to be pointer targets, but disallow assignments to those pointers; those constraints are even more complex than the current ones.)

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 (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.

Constraint 4 (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 (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 through 8 (constraints on explicit mapping statements) protect dummy and global data objects from realignment and redistribution (another type of side effect). In addition, constraint 5 restricts explicit declaration of the mapping of dummy arguments and local variables. This is because the function may be invoked concurrently, with each invocation operating on a segment of data whose distribution is specific to that invocation. Thus, the distribution of a dummy object may be inherited from the corresponding actual argument. For similar reasons, local variables may be aligned with dummy arguments (either directly or through other local variables), but may not have arbitrary mappings.

The last constraint prevents I/O, whose order would be non-deterministic in the context of concurrent execution. A `PAUSE` statement requires input and so is disallowed for the same reason. Finally, a `STOP` brings execution to a halt, which is a rather drastic side effect.

Pure subroutine definition

To define pure subroutines, Fortran 90 Rule R1219 is changed to:

subroutine-subprogram **is** *subroutine-stmt*
  [*pure-directive*]
  [*specification-part*]
  [*execution-part*]
  [*internal-subprogram-part*]
  end-subroutine-stmt

with the following constraints in addition to those in Section 12.5.2.3 of the Fortran 90 standard:

Constraint: If a *procedure-name* is present in the *pure-directive*, it must match the *subroutine-name* in the *subroutine-stmt*.

Constraint: The *specification-part* of a pure subroutine must specify the intents of all non-pointer and non-procedure dummy arguments.

Constraint: In a pure subroutine, a local variable must not have the `SAVE` attribute. (Note that this means that a local variable cannot be initialized in a *type-declaration-stmt* or a *data-stmt*, which imply the `SAVE` attribute.)

Constraint: 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*;
- 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 `(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: In a pure subroutine, a dummy argument must not appear in an explicit mapping directive unless it has 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 *pause-stmt*, *stop-stmt* or I/O statement (including a file operation).

To declare that a subroutine is pure, a *pure-directive* must be given.

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

Pure procedure interfaces

To define interface specifications for pure procedures, Fortran 90 Rule R1204 is changed to:

interface-body **is** *function-stmt*
  [*pure-directive*]
  [*specification-part*]
  end-function-stmt
or *subroutine-stmt*
  [*pure-directive*]
  [*specification-part*]
  end-subroutine-stmt

with the following constraint in addition to those in Section 12.3.2.1 of the Fortran 90 standard:

Constraint: An *interface-body* of a pure subroutine must specify the intents of all non-pointer and non-procedure dummy 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 block, but this is not required.

- A procedure that is not declared pure at its definition must not be declared pure in an interface block.

That is, if an interface body contains a *pure-directive*, then the corresponding procedure definition must also contain it, though the reverse is not true. When a procedure definition with a *pure-directive* 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 Section 12.4.1 of the Fortran 90 standard:

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

4.3.3 Examples of pure procedure usage

Pure functions may be used in expressions in `FORALL` statements and constructs, unlike general functions. Because a *forall-assignment* may be an array assignment, the pure function can have an array result. For example:

```
INTERFACE
  FUNCTION f(x)
    !HPF$ PURE f
    REAL, DIMENSION(3) :: f, 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))
```

Such functions may be particularly helpful for performing row-wise or column-wise operations on an array.

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, for example in

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

REAL a(n)
```

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

This may sometimes provide an alternative to using sequences of masked FORALL statements with their potential synchronization overhead.

Because pure procedure 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 following fragment performs an iterative algorithm on every element of an array:

```
FUNCTION iter(x)
  !HPF$ PURE iter
  COMPLEX x, xtmp
  INTEGER iter, i
  i = 0
  xtmp = -x
  DO WHILE (ABS(xtmp).LT.2.0 .AND. i.LT.1000)
    xtmp = xtmp * xtmp - x
    i = i + 1
  ENDDO
  iter = i
END FUNCTION

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

Note that different amounts of computation may be required for different inputs. Some machines may not be able to take advantage of this flexibility.

4.3.4 Comments on Pure Procedures

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 referenced as 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 dummy argument (in the case of a function), or perform any I/O or STOP operation. Note the use of the word *conceivably*; it is not sufficient for a pure function merely to be side-effect free *in practice* (e.g., 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 some functions of this nature is unavoidable if strict compile-time checking is to be used, due to the undecidability of the halting problem. In the choice between compile-time checking and flexibility, the HPF committee decided in favor of enhanced checking.

It is expected that most HPF 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 (if they are functions) 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 for statically checkable side-effect freedom and the elimination 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.

4.4 The INDEPENDENT Directive

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

The syntax of the INDEPENDENT directive is

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

Constraint: An *independent-directive* must immediately precede a DO or FORALL statement.

Constraint: If the *integer-variable-list* is present in an *independent-directive*, then the variables named must be the index variables of a set of perfectly nested DO loops or index-names from the same FORALL header. A set of loops is perfectly nested if each loop (except the innermost loop) encloses one other loop in the set directly and no other statements.

Constraint: The NEW option may be used only with perfectly nested loops.

Constraint: A variable named in theNEW option or any component or element thereof must not:

- Be a pointer, dummy argument, common block, module, program unit or entry point;
- Have the SAVE or TARGET attribute; nor
- Be associated with any data object that is used in the execution part of the nest of DO loops and not named in this NEW option.

The directive is said to apply to the indexes named in its *integer-variable-list*, or equivalently to the loops or FORALL indexed by those variables. If no *integer-variable-list* is present, then it is as if it were present and contained the index variable for the `DO` or `FORALL` immediately following the directive.

When applied to a nest of `DO` loops, an `INDEPENDENT` directive is an assertion by the programmer that no iteration can affect any other iteration, either directly or indirectly. Any change of state to data used by an iteration is considered to affect that iteration, even if the results are mathematically equivalent. Thus, an `INDEPENDENT` loop can be executed safely in parallel if computation resources are available. The directive is purely advisory and a compiler is free to ignore it if it cannot make use of the information. The following are some consequences of the `INDEPENDENT` assertion:

- No iteration assigns to any *atomic data object* which is accessed (read or written) by another iteration. (An atomic data object is a Fortran 90 data object that has no subobjects.)
- There are no exits from any loop in the nest other than normal termination.
- No data realignment or redistribution is performed in the loop.

The `NEW` option specifies that the assertion by the `INDEPENDENT` directive would be correct *if* distinct storage units are used for the named variables for each iteration of the `DO` loop nest. 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 become undefined immediately before execution of every iteration of the loop nest, and also become undefined immediately after the completion of each iteration of the loop nest. (This 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.)

For example:

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

asserts that the array `p` does not have any repeated entries (else they would cause interference when `a` was assigned). It also limits how `a` and `b` may be storage associated. (The remaining examples in this section assume that no variables are storage or sequence associated.)

Another example:

```
!HPF$ INDEPENDENT (i1,i2,i3)
DO i1 = 1,n1
 DO i2 = 1,n2
  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. It is not relevant that the outer loops read the same elements from b and c, because those arrays are not assigned.

Another example:

```
!HPF$ INDEPENDENT (i,j), NEW(vl, vr, ul, ur)
DO i = 1, n, 2
  DO j = 1 , n, 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 this loop would not 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.

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 the form of a FORALL 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. In the case of a FORALL, any of the variables may be mentioned in the INDEPENDENT directive:

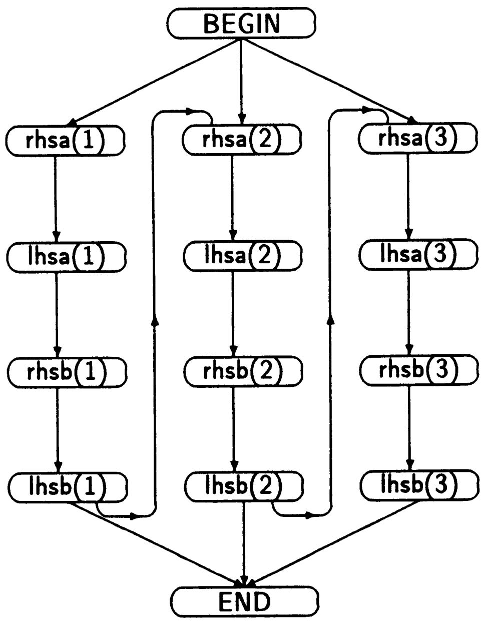
```
!HPF$ INDEPENDENT (i1,i3)
FORALL ( i1=2:n1-1, i2=2:n2-1, i3=2:n3-1 )
  a(i1,i2,i3) = a(i1,i2-1,i3)
END FORALL
```

This means that for any given values for i1 and i3, all the right-hand sides for all values of i2 must be computed before any assignment are done for that specific pair of (i1,i3) values; but assignments for one pair of (i1,i3) values need not wait for evaluation of the right-hand side for a different pair of (i1,i3) values.

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 an a FORALL. (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 nodel; 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

```
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
```



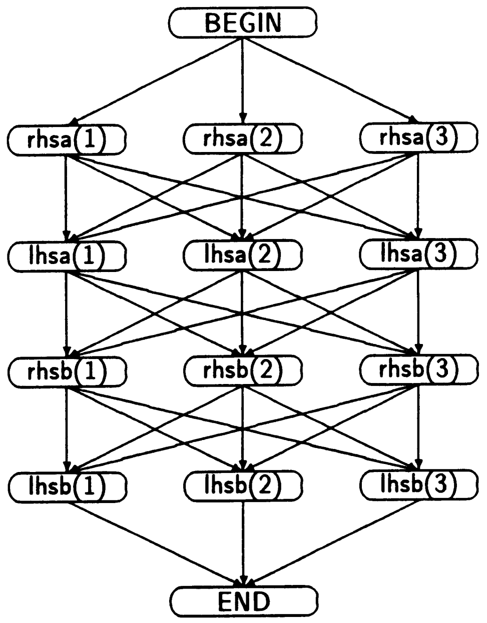
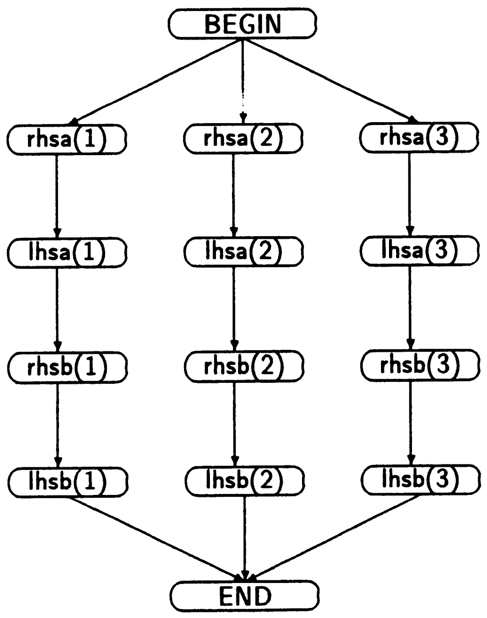


Figure 4.1: Dependences in DO and FORALL without INDEPENDENT assertions

```
!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
```



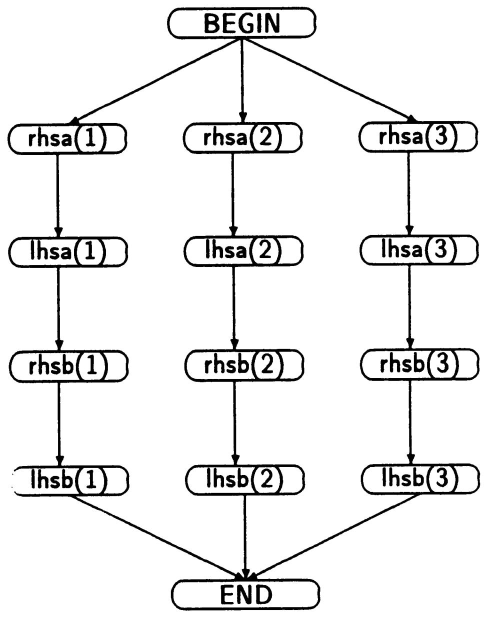


Figure 4.2: Dependences in DO and FORALL with INDEPENDENT assertions

dependences that a compiler need enforce are those in Figure 4.2. That is, the programmer 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 standard conforming, is not defined by HPF, and the compiler may take any action it deems appropriate.

Chapter 5

Intrinsic and Library Procedures

HPF includes Fortran 90's intrinsic procedures. It also adds a number of new intrinsic procedures in three categories: system inquiry intrinsic functions, mapping inquiry intrinsic subroutines, 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 procedures, HPF defines a library module, HPF_LIB, that must be provided by vendors of any full HPF implementation.

5.1 System Inquiry Intrinsic Functions

In a multi-processor implementation, the processors may be arranged in an implementation-dependent n-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. Accordingly, 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.

5.1.1 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 ($1 \leq$ DIM $\leq$ n), where n is the rank of the processor-dependent hardware processor array or, if DIM

is absent, the total number of elements, equ to or greater than one, of the processor-dependent hardware processor array.

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_PROCESSORS(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, and
- the value of NUMBER_OF_PROCESSORS(DIM=1) is 1.

5.1.2 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: For a computer with 8192 processors arranged in a 128 by 64 rectangular grid, the value of PROCESSORS_SHAPE() is (/ 128,64 /). For a computer with 8192 processors arranged in a hypercube, the value of PROCESSORS_SHAPE() might be (/ 2,2,2,2,2,2,2,2,2,2,2,2,2 /). For a single processor workstation, the value of PROCESSORS_SHAPE() is (/ 1 /).

5.1.3 Discussion and Pragmatic Usage

The values of system inquiry functions are always restricted expressions; thus they may be used in specification expressions. They may not, however, 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.

References to system inquiry functions may occur in HPF directives, as in:

```
!HPF$ TEMPLATE T(100, 3*NUMBER_OF_PROCESSORS())
```

The definition of NUMBER_OF_PROCESSORS is modeled on the definition of the SIZE intrinsic function.

The definition of PROCESSORS_SHAPE is modeled on the definition of the SHAPE intrinsic function.

The rank of the processor array is returned by

```
   SIZE(PROCESSORS_SHAPE())
```

an expression that may occur in any specification expression.

As a result of being a restricted expression, suitably constrained references to system inquiry functions may occur in specification expressions as, for example, lower or upper bounds of an *explicit-shape-spec* of an *array-spec* in a *type-declaration-stmt*, as in:

```
      INTEGER, DIMENSION(SIZE(PROCESSORS_SHAPE())) :: PS
      PS = PROCESSORS_SHAPE()
!     PS(2) = NUMBER_OF_PROCESSORS(DIM=2)
```

5.2 Computational Intrinsic Functions

This section extends the set of Fortran 90 computational intrinsic functions and generalizes some Fortran 90 intrinsic functions.

5.2.1 Extension to MAXLOC and MINLOC

The MAXLOC and MINLOC intrinsic functions are redefined to have a third, optional DIM argument that works exactly as does the DIM argument of MAXVAL. If such an argument is present, then the shape of the result equals the shape of the first argument with one dimension (the one indicated by the DIM argument) deleted; it is as if a series of rank-one MAXLOC or MINLOC operations were performed. The rank of the result is one less than the rank of the first argument. If the smallest (MINLOC) or largest (MAXLOC) element along a given dimension is not unique, then the location of the first one is returned. The declared lower bounds of the input array play no role in determining the output: The values returned by MAXLOC and MINLOC are computed as if the lower bounds of all dimensions were 1. The optional MASK argument is retained and may be used together with the DIM argument.

Note that the behavior of MAXLOC and MINLOC without the DIM argument is quite different. In this case, a rank-one integer array of size equal to the rank of ARRAY is returned, giving the subscripts of the first element in array element order with the smallest (MINLOC) or largest (MAXLOC) value.

Thus, if A has DIMENSION(3,4), then

```
SHAPE(MAXLOC(A))       has the value (/ 2 /)
SHAPE(MAXLOC(A,DIM=1)) has the value (/ 4 /)
SHAPE(MAXLOC(A,DIM=2)) has the value (/ 3 /).
```

Example: If A has the value

$$\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 0 & 4 & 6 & -4 \end{bmatrix}$$

then

```
MINLOC(A)       has the value (/ 1,2 /)
MAXLOC(A)       has the value (/ 1,3 /)
MINLOC(A, DIM=1) has the value (/ 1,1,2,3 /)
MAXLOC(A, DIM=1) has the value (/ 2,2,1,2 /)
MINLOC(A, DIM=2) has the value (/ 2,3,4 /)
MAXLOC(A, DIM=2) has the value (/ 3,2,3 /).
```

5.2.2 ILEN

An elemental, integer length intrinsic function. Its action on a scalar is:

```
    ILEN(X) = ceiling(log2( IF x < 0 THEN -x ELSE x+1 ))
```

ILEN(x) is the number of bits required to store a 2's-complement signed integer x. As examples of its use, 2**ILEN(N-1) rounds N up to a power of 2 (for $N > 0$), whereas 2**(ILEN(N)-1) rounds N down to a power of 2.

Note that a given integer value will always produce the same result from ILEN, without regard to the number of bits in the representation of the integer. That is because ILEN counts bits from the least significant bit.

Argument. X must be integer. It may be scalar or array valued.

Result shape, type, and type parameters. Same as X.

As an elemental, integer-valued intrinsic, ILEN may appear in a specification expression.

5.3 Mapping Inquiry Intrinsic Subroutines

HPF provides a rich set of data mapping directives. These directives are advisory in nature. The mapping inquiry intrinsic 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 a non-HPF subprogram is invoked. For these reasons, HPF includes mapping inquiry intrinsic 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 intrinsic subroutines with optional arguments.

5.3.1 Alignment Inquiry Subroutine

```
SUBROUTINE HPF_ALIGNMENT(ARRAY, LB, UB, STRIDE, AXIS_MAP,    &
        IDENTITY_MAP, DYNAMIC, NCOPIES)
INTEGER, OPTIONAL, INTENT(OUT), DIMENSION(:) ::  &
        LB, UB, STRIDE, AXIS_MAP
INTEGER, OPTIONAL, INTENT(OUT) :: NCOPIES
LOGICAL, OPTIONAL, INTENT(OUT) :: IDENTITY_MAP, DYNAMIC
```

Mandatory ARRAY

Optional LB, UB, STRIDE, AXIS_MAP, IDENTITY_MAP, DYNAMIC, NCOPIES

The `HPF_ALIGNMENT` subroutine returns information regarding the correspondence of an array and the entity (array or template) to which it is ultimately aligned. `ARRAY` is the only INTENT(IN) argument; all the remaining arguments are optional, INTENT(OUT).

ARRAY The array about which alignment information is requested. ARRAY may not be a pointer that is disassociated or an allocatable array that is not allocated.

LB An integer array. The first element of the ith axis of ARRAY is ultimately aligned to the LB(i)th align-target element along the axis of the align-target associated with the ith axis of ARRAY.

UB An integer array. The last element of the ith axis of ARRAY is ultimately aligned to the UB(i)th align-target element along the axis of the align-target associated with the ith axis of ARRAY.

STRIDE An integer array The ith element of STRIDE contains the stride used in aligning the elements of ARRAY along its ith axis.

AXIS_MAP An integer array. The ith element of AXIS_MAP contains the alignee axis associated with the ith array axis. If AXIS_MAP is 0, the axis is a collapsed axis.

IDENTITY_MAP A logical scalar variable that will be true if the ultimate align-target associated with ARRAY has a shape identical to ARRAY, the axes are mapped using the identity permutation, and the strides are all positive. if an array is ultimately aligned to itself, then IDENTITY_MAP has a .TRUE. value.

DYNAMIC A logical scalar variable that will be true if ARRAY has the DYNAMIC attribute.

NCOPIES An integer scalar variable equal to the product of the extents of all align-target axes over which ARRAY has been replicated. For a non-replicated array, for example, this will be 1.

5.3.2 Template Inquiry Subroutine

```
SUBROUTINE HPF_TEMPLATE(ARRAY, TEMPLATE_RANK, LB, UB, AXIS_TYPE,   &
        AXIS_INFO, NUMBER_ALIGNED, DYNAMIC)
INTEGER, OPTIONAL, INTENT(OUT), DIMENSION(:) :: LB, UB, AXIS_INFO
CHARACTER*(*), OPTIONAL, INTENT(OUT) :: AXIS_TYPE(:)
INTEGER, OPTIONAL, INTENT(OUT) :: NUMBER_ALIGNED, TEMPLATE_RANK
LOGICAL, OPTIONAL, INTENT(OUT) :: DYNAMIC
```

Mandatory ARRAY

Optional LB, UB, AXIS_TYPE, AXIS_INFO, NUMBER_ALIGNED, TEMPLATE_RANK, DYNAMIC

The HPF_TEMPLATE subroutine returns information regarding the ultimate align-target associated with an array. The main difference between HPF_TEMPLATE and HPF_ALIGNMENT is that the former returns information concerning the array from the template's point of view (assuming the alignment is to a template rather than to an array), while the latter returns information from the array's point of view. ARRAY is the only INTENT(IN) argument; all the remaining arguments are optional, INTENT(OUT).
Result Shape. Array outputs are rank one and of size equal to the rank of the template, which is returned in TEMPLATE_RANK.

ARRAY The array about which ultimate align-target information is requested. ARRAY may not be a pointer that is disassociated or an allocatable array that is not allocated.

TEMPLATE_RANK An integer scalar variable giving the number of axes in the ultimate align-target. This can be different than the number of ARRAY axes, due to collapsing and replicating.

LB An integer array. The ith element of LB contains the declared align-target lower bound for the ith template axis.

UB An integer array. The ith element of UB contains the declared align-target upper bound for the ith template axis.

AXIS_TYPE A rank one array of type default character, of length at least 10; element (i) of the result returns information about the ith axis of the align-target. The following values are defined by HPF (implementations may define other values):

> **'NORMAL'** The axis has an axis of `ARRAY` aligned to to it. `AXIS_INFO` contains the axis of `ARRAY` aligned with the axis of the align-target.
>
> **'SINGLE'** `ARRAY` is aligned with one coordinate of the align-target axis. `AXIS_INFO` contains the coordinate to which `ARRAY` is aligned.
>
> **'REPLICATED'** `ARRAY` is replicated along this align-target axis. `AXIS_INFO` contains the number of copies of `ARRAY` along the axis. This is an implementation-specific quantity.

AXIS_INFO See the desciption of `AXIS_TYPE` above.

Example: Given

```
        REAL A(4, 20)
!HPF$   TEMPLATE T(30, 150, 8, 200)
        INTEGER AINFO(4)
        CHARACTER*10 ATYPE(4)
!HPF$   ALIGN A(I,J,*) WITH T(J+5, 100, 10-2*I, *)
        CALL HPF_TEMPLATE(A, AXIS_TYPE=ATYPE, AXIS_INFO=AINFO)
```

then

```
ATYPE = (/'NORMAL', 'SINGLE', 'NORMAL', 'REPLICATED'/)
```

and

```
AINFO = (/2, 100, 1, 200/)
```

NUMBER_ALIGNED An integer scalar variable giving the total number of arrays aligned to the ultimate align-target. This is the number of arrays that will be moved when the align-target is redistributed.

DYNAMIC A logical scalar variable that will be true if the align-target is dynamic.

5.3.3 Distribution Inquiry Subroutine

```
SUBROUTINE HPF_DISTRIBUTION(ARRAY, AXIS_TYPE, AXIS_INFO,    &
        PROCESSORS_RANK, PROCESSORS_SHAPE)
INTEGER, OPTIONAL, INTENT(OUT) :: AXIS_INFO(:), PROCESSORS_RANK
CHARACTER*(*), OPTIONAL, INTENT(OUT) :: AXIS_TYPE(:)
INTEGER, OPTIONAL, INTENT(OUT) :: PROCESSORS_SHAPE(:)
```

Mandatory ARRAY

Optional AXIS_TYPE, AXIS_INFO, PROCESSORS_SHAPE, PROCESSORS_RANK

The `HPF_DISTRIBUTION` subroutine returns information regarding the distribution of the ultimate align-target associated with an array. `ARRAY` is the only INTENT(IN) argument; all the remaining arguments are optional, INTENT(OUT).

ARRAY The array for which ultimate align-target distribution information is requested. ARRAY may not be a pointer that is disassociated or an allocatable array that is not allocated.

AXIS_TYPE A rank one array of type default character and of length at least 9, and size equal to the rank of the align-target of `ARRAY` (which is returned by HPF_TEMPLATE in TEMPLATE_RANK), that returns information about the distribution of each axis of the align-target. The following values are defined by HPF (implementations may define other values):

'BLOCK' The axis is distributed BLOCK. `AXIS_INFO` contains the block size.

'CYCLIC' The axis is distributed CYCLIC. `AXIS_INFO` contains the block size.

'COLLAPSED' The axis is collapsed (distributed with the "*" specification)

AXIS_INFO A rank one integer array. See the desciption of `AXIS_TYPE` above.

PROCESSORS_RANK An integer scalar variable giving the rank of the processor arrangement onto which `ARRAY` is distributed.

PROCESSORS_SHAPE An array of type default integer and of size equal to the value returned in PROCESSORS_RANK, giving the shape of the processor arrangement onto which `ARRAY` 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.

5.3.4 Examples

Consider the declarations below:

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

The results of `HPF_ALIGNMENT` will be:

	A	B	C
LB	4, 2, ...	1, 1, ...	1, N/A, 1, ...
UB	31, 20, ...	20, 30, ...	20, N/A, 10, ...
STRIDE	3, 2, ...	1, 1, ...	-1, N/A, 1, ...
AXIS_MAP	1, 2, ...	1, 2, ...	2, 0, 1, ...
IDENTITY_MAP	.FALSE.	.TRUE.	.FALSE.
DYNAMIC	.TRUE.	.FALSE.	.FALSE.
NCOPIES	1	1	1

and the result of `HPF_TEMPLATE` will be

	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, ...
NUM.AL.	3	3	3
TEMP. RANK	2	2	2
DYNAMIC	.FALSE.	.FALSE.	.FALSE.

Finally `HPF_DISTRIBUTION` will produce

	A	B
AXIS_TYPE	'BLOCK', 'BLOCK', ...	'CYCLIC', 'BLOCK', ...
AXIS_INFO	10, 10, ...	1, 15, ...
PROCESSORS_SHAPE	4, 2, ...	4, 2, ...
PROCESSORS_RANK	2	2

Note that the values of the block sizes (in `AXIS_INFO`) are not specified by HPF, but may be implementation-dependent.

5.4 Computational Library Functions

This section consists of five groups of computational library functions, to be available in the HPF library module, HPF_LIB. Use of these functions must be accompanied by an appropriate USE statement in each scoping unit in which they are used. They are not intrinsic. Thus, they are not allowed in specification expressions.

5.4.1 New Reduction Functions

Just as Fortran 90 has the correspondences:

operator/intrinsic	reduction intrinsic
`+`	`SUM, COUNT`
`*`	`PRODUCT`
`.AND.`	`ALL`
`.OR.`	`ANY`
`MAX`	`MAXVAL`
`MIN`	`MINVAL`

it is useful to have reduction versions of certain other operators and intrinsic functions in the language that happen to be associative and commutative. Therefore the new functions IALL, IANY, IPARITY, and PARITY are defined.

operator/intrinsic	reduction function
IAND	IALL
IOR	IANY
IEOR	IPARITY
.NEQV.	PARITY

These reductions have the same argument lists (including optional DIM and MASK arguments) as the Fortran 90 reductions.

```
    IALL( (/ 7,3,10 /) )  yields 2
    IANY( (/ 7,3,10 /) )  yields 15
    IPARITY( (/ 7,3,10 /) )  yields 14

  LOGICAL T,F
  PARAMETER (T = .TRUE., F = .FALSE.)      !just for conciseness

  PARITY( (/ T,F,F,T,T,F,F,F,T,T /) )  yields .TRUE.
  PARITY( (/ T,F,F,T,T,F,F,F,T,F /) )  yields .FALSE.
```

Some of these are particularly valuable when used with the corresponding prefix functions, Section 5.4.3.

The identity element for the reduction PARITY is false, for the reductions IANY and IPARITY is zero, and for the reduction IALL is -1 (in twos-complement). COUNT does not have an identity, as it maps logicals to integers and returns zero if there are no true values to be counted. The identities for the other reductions are defined in the Fortran 90 standard.

5.4.2 Combining-Scatter Functions

For every reduction operation XXX in the language, HPF introduces a new function:

```
XXX_SCATTER(SOURCE,BASE,IDX1,..., IDXn, MASK)
```

The IDX arguments are integer arrays. The number of IDX arguments must equal the rank of BASE. The SOURCE, MASK (if present), and all the IDX arguments must be conformable. BASE must have rank m, where m is the number of IDX arrays actually present. The result delivered by the function is conformable with BASE. The type and type parameters of SOURCE and BASE must be the same (exception: COUNT_SCATTER), and the result has the type and type parameters of BASE. The allowed types are:

XXX	Allowed Types
SUM	Real, Complex, Integer
COUNT	BASE = Integer, SOURCE = Logical

PRODUCT	Real, Complex, Integer
MAXVAL	Real, Integer
MINVAL	Real, Integer
IALL	Integer
IANY	Integer
IPARITY	Integer
ALL	Logical
ANY	Logical
PARITY	Logical

Since SOURCE and all the IDX arrays are conformable, for every element s in SOURCE there is a corresponding element in each of the IDX arrays. Let $i1$ be the value of the element of IDX1 that is indexed by the same subscripts as element s of SOURCE. More generally, for each $j = 1, 2, ..., n$, let ij be the value of the element of $IDXj$ that corresponds to element s in SOURCE, where n is the rank of BASE. The integers $ij, j = 1, ..., n$, form a subscript selecting an element of BASE: $BASE(i1, i2, ..., in)$.

Thus SOURCE and the IDX arrays establish a mapping from all the elements of SOURCE 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 SOURCE.

Since BASE and the result of XXX_SCATTER are conformable, there is a corresponding element of the result for each element of BASE.

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. Let the elements of S be $s1, ..., sm$. Let @ denote an infix form of operation XXX. The element of the result corresponding to the element b of BASE is the result of evaluating $s1@s2@...@sm@b$ or any mathematically equivalent expression (as defined in Section 7.1.7.3 of the Fortran 90 standard).

Thus, the order of operations is arbitrary, and may differ on two otherwise identical runs of the same HPF program. This matters when the combining operation is not both associative and commutative, for example floating-point addition. In fact, because machine arithmetic is not associative (not even fixed-point, because of overflow) the programmer must be sure that the nondeterministic order of evaluation of the result will not produce undesirable effects.

If the optional argument MASK is present, then only the elements of SOURCE in positions for which MASK is true participate in the operation. All other elements of SOURCE and of the IDX arrays are ignored.

Thus the result of the expression

```
SUM_SCATTER(SOURCE,BASE,IDX1,IDX2,...,IDXn,MASK)
```

could be computed as

```
result = BASE
DO J1=LBOUND(SOURCE,1),UBOUND(SOURCE,1)
  DO J2=LBOUND(SOURCE,2),UBOUND(SOURCE,2)
    ...
      DO Jk=LBOUND(SOURCE,k),UBOUND(SOURCE,k)
```

```
                IF (MASK(J1,J2,...,Jk))          &
                   result(IDX1(J1,J2,...,Jk),    &
                          IDX2(J1,J2,...,Jk),    &
                          ...                    &
                          IDXn(J1,J2,...,Jk)) =  &
                   result(IDX1(J1,J2,...,Jk),    &
                          IDX2(J1,J2,...,Jk),    &
                          ...                    &
                          IDXn(J1,J2,...,Jk)) + SOURCE(J1,J2,...,Jk)
              END DO
             ...
           END DO
         END DO
```

where k is the rank of SOURCE. (However, this nest of DO loops makes a greater commitment to the particular order in which the combining operations are carried out than the order—namely, none!— guaranteed by the XXX_SCATTER function.)

In addition, COPY_SCATTER is the combining-send function generated by the (non-commutative) binary operator

```
COPY_operation(x,y) = x
```

When COPY_SCATTER combines source elements $s1, s2, ..., sm$ with base element b, the processor may apply the operations in the expression $s1@s2@...@sm@b$ in any order it chooses, not defined by HPF (where $x@y$ denotes COPY_operation(x,y)). Since b occurs on the right in the expression, the processor cannot select b as the result, because COPY_operation selects its first argument as its result. Thus an element of the result delivered by

```
COPY_SCATTER(SOURCE, BASE, IDX1, ..., IDXn)
```

corresponding with an element of BASE that is associated with a non-empty set from SOURCE has the same value as *some* SOURCE element from that set. So if multiple elements of SOURCE are sent to the same result element, one of them will be assigned and the rest, as well as the corresponding element of BASE, will be effectively discarded.

Example:

```
A = (/ 10., 20., 30., 40., -10./)
X = (/ 1.,  2.,  3.,  4./)
V = (/ 3,   2,   2,   1,   1/)
X = SUM_SCATTER(A,X,V, MASK=(A > 0) )
```

yields the result X = (/41., 52., 13., 4./).

If all elements of V were distinct, one could write this in Fortran 90 as

```
X(V) = X(V) + MERGE(A, 0., A > 0.)
```

The function SUM_SCATTER is applicable even if V contains duplicate values. Note that the rank-two case

```
      X(V,W) = X(V,W) + B
```

must be rendered using SPREAD:

```
      X = SUM_SCATTER(B,X,SPREAD(V,DIM=2,NCOPIES=SIZE(X,2)),  &
                          SPREAD(W,DIM=1,NCOPIES=SIZE(X,1)))
```

in order to duplicate the cross-product effect of ordinary array subscripting. (This definition of XXX_SCATTER does *not* perform such a cross product of indices because it is more general and in practice more useful without the cross-product effect built in.)

When scatter along one or more axes of a multidimensional array is required, a surrounding FORALL may be used. For example, the idiom used to SUM_SCATTER the (j,k) planes of an (i,j,k)-indexed rank-three array, using the rank-one index vector V is

```
      REAL, ARRAY(NI, NJ, NK) :: SRC, DEST
      LOGICAL MASK(NI, NJ, NK)
      INTEGER V(NI)
      FORALL (J = 1:NJ, K = 1:NK)                                &
         DEST(:, J, K) = SUM_SCATTER(SRC(:,J,K), DEST(:,J,K),    &
              V, MASK(:, J, K))
```

which has the same effect as

```
      DO I = 1, NI
        WHERE (MASK(I, :, :))   &
          DEST(V(I), :, :) = DEST(V(I), :, :) + SRC(I, :, :)
      ENDDO
```

but may be more efficient, and makes no guarantees as to the order of evaluation.

5.4.3 Prefix and Sufix Functions

For every reduction operation XXX in the language, HPF introduces the two new functions XXX_PREFIX and XXX_SUFFIX. They take the same arguments as the corresponding reduction function, (an array of appropriate type, an optional scalar integer DIM argument, an optional, logical array argument MASK conformable with ARRAY) plus two additional optional arguments:

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

Each element of the result is the reduction under the operator XXX of a (possibly empty) set of elements of ARRAY.

Example:

```
      MAXVAL_PREFIX((/ 3, 2, 4, 1, 2/))
  is
                    (/ 3, 3, 4, 4, 4/).

      MAXVAL_SUFFIX((/ 3, 2, 4, 1, 2/))
  is
                    (/ 4, 4, 4, 2, 2/).
```

The value of these functions is conformable with ARRAY.

The result has the same type and type parameters as ARRAY, with the exception of COUNT_PREFIX and COUNT_SUFFIX which take a logical array argument and return an integer array result. The allowed operations and the corresponding allowed types for ARRAY are given in the table below.

XXX	Allowed Types
SUM	Real, Complex, Integer
COUNT	Result = Integer, ARRAY = Logical
PRODUCT	Real, Complex, Integer
MAXVAL	Real, Integer
MINVAL	Real, Integer
IALL	Integer
IANY	Integer
IPARITY	Integer
ALL	Logical
ANY	Logical
PARITY	Logical

If the DIM argument is omitted, then the arrays are processed in array element order ("column-major"), as if temporarily regarded as rank-one. If it is present, then it must be an integer scalar between one and the rank of ARRAY. In this case, completely independent prefix or suffix operations occur along the selected dimension of ARRAY.

Example: If A has the value

$$\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 0 & 4 & 6 & -4 \end{bmatrix}$$

then SUM_PREFIX(A) has the value

$$\begin{bmatrix} 0 & -2 & 14 & 16 \\ 3 & 2 & 13 & 18 \\ 3 & 6 & 19 & 14 \end{bmatrix}$$

SUM_PREFIX(A, DIM=1) has the value

$$\begin{bmatrix} 0 & -5 & 8 & -3 \\ 3 & -1 & 7 & -1 \\ 3 & 3 & 13 & -5 \end{bmatrix}$$

SUM_PREFIX(A, DIM=2) has the value

$$\begin{bmatrix} 0 & -5 & 3 & 0 \\ 3 & 7 & 6 & 8 \\ 0 & 4 & 10 & 6 \end{bmatrix}$$

Array elements corresponding to positions where the MASK is false do not contribute to the running accumulation. However, the result is still defined for corresponding positions in the result.

Example:

```
      MAXVAL_PREFIX( (/ 3, 2, 4, 5, 6/),     &
              MASK  = (/ T, F, F, T, F/))
  has the value
                      (/ 3, 3, 3, 5, 5/)
```

In actual practice, results may not be required in those positions; in such cases the programmer may be able to use the WHERE statement to inform the compiler:

```
      WHERE (FOO) A=SUM_PREFIX(B,MASK=FOO)
```

The first additional optional argument is called SEGMENT, which is of type logical and conformable with the ARRAY argument. If present, the array is divided into pieces corresponding to contiguous sequences of true or false elements of SEGMENT. The beginning of a piece is a place where the running accumulation is to be reset before processing the corresponding array element.

Example:

```
  LOGICAL T,F
  PARAMETER (T = .TRUE., F = .FALSE.)

  MAXVAL_PREFIX((/ 3, 2, 4, 1, 6/),  &
        SEGMENT=(/ T, T, T, F, F/))  yields  (/ 3, 3, 4, 1, 6/).
                 -------  ----                -------  ----
              two input segments         two independent results
```

The second additional optional argument, a scalar logical, is called EXCLUSIVE, default value false, which determines whether the prefix or suffix operation is inclusive (the default) or exclusive. (The inclusive sum-prefix of (/ 1,2,3,4 /) is (/ 1,3,6,10 /) whereas the exclusive sum-prefix is (/ 0,1,3,6 /).)

In every case, every element of the result has a value equal to the reduction of certain selected elements of ARRAY, or an identity value (zero for SUM_PREFIX or SUM_SUFFIX, for example) if no elements of ARRAY are selected for that result element. 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.

The identity element for the reduction PARITY is false, for the reductions IANY and IPARITY is zero, and for the reduction IALL is -1 (assuming twos-complement). COUNT does not have an identity, as it maps logicals to integers and returns zero if there are no true values to be counted. The identities for the other reductions are defined in the Fortran 90 standard.

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.

For xxx_PREFIX, no element that follows A in the array element ordering of ARRAY contributes to R. For xxx_SUFFIX, no element that precedes A in the array element ordering of ARRAY contributes to R. This rule applies even when the DIM argument is present, since array element order increases with an increase in any component of an array element index.

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.

If the MASK argument is provided, an element Z of ARRAY does not contribute to R if the element of MASK corresponding to Z is false.

If the SEGMENT argument is provided, an element Z of ARRAY does not contribute unless the elements B and Y of SEGMENT corresponding to A and Z (respectively), and the intervening elements of SEGMENT as well, all have the same value. If the DIM argument is not present, then the "intervening" elements are all elements between them in array element order; if the DIM argument is present, then the "intervening" elements are those having indices the same as those of both B and Y, except the index for dimension DIM, which must be between (and possibly equaling) the indices of B and Y for dimension DIM. Thus, the prefix or suffix operation is performed on groups of elements of ARRAY, where a group corresponds to a maximal contiguous run of like-valued elements of SEGMENT.

If the SEGMENT argument is omitted, then the result is computed using a default SEGMENT all elements of which are true. Thus, without the DIM argument, there is exactly one group, while if DIM is present, there is one group for each valid set of indices of ARRAY other than the index selected by DIM.

If the EXCLUSIVE argument is provided and is true, then A itself does not contribute to R.

In addition, the operation COPY_PREFIX replicates the first (lowest-indexed) element of each segment throughout the segment, and the operation COPY_SUFFIX replicates the last (highest-indexed) element of each segment throughout the segment.

Examples:

```
(a) SUM_PREFIX( (/1,3,5,7/) ) yields (/1,4,9,16/)
(b) SUM_SUFFIX( (/1,3,5,7/) ) yields (/16,15,12,7/)

    LOGICAL T,F
    PARAMETER (T = .TRUE., F = .FALSE.)

(c) COUNT_PREFIX( (/T,F,F,T,T,T,F,T,F/) )
                                !yields (/1,1,1,2,3,4,4,5,5/)
(d) COUNT_PREFIX( (/T,F,F,T,T,T,F,T,F/), EXCLUSIVE=T)
                                !yields (/0,1,1,1,2,3,4,4,5/)

(e) SUM_PREFIX( (/1,2,3,4,5,6,7,8,9/),  &
        SEGMENT=(/T,T,T,T,F,F,T,F,F/))
                 ------- --- - ---
                 four input segments

yields

               (/1,3,6,10,5,11,7,8,17/)
                 -------- ---  - ----
                 four independent result segments

(f) COPY_PREFIX( (/1,2,3,4,5,6,7,8,9/),  &
         SEGMENT=(/T,T,T,T,F,F,T,F,F/))
                  ------- --- - ---
                  four input segments
```

```
yields

          (/1,1,1,1,5,5,7,8,8/)
            ------- --- - ---
            four independent result segments
```

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
```

Note: Connection Machine software delimits the segments by indicating the *start* of each segment. Cray MPP Fortran 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 parallel prefix and parallel suffix without changing its interpretation (start versus stop).

(B) It seems to be equally inconvenient for every existing architecture! However, it is not that hard to accommodate.

(C) The start-bit or stop-bit representation is easily converted to this form by using PARITY_PREFIX or PARITY_SUFFIX.

Examples:

```
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))
```

These might be standard idioms for a compiler to recognize.

5.4.4 Sorting Functions

This section introduces two sorting functions, GRADE_UP and GRADE_DOWN.

```
GRADE_UP(ARRAY,DIM)
```

The argument ARRAY may be of type integer, real, or character.

The result is an integer array with shape as explained below.

If the optional DIM argument is present, then the result has the same shape as the ARRAY. Suppose DIM has the value k; then the result R has the property that if one computes the array

```
B(i1,i2,...,ik,...,in)=ARRAY(i1,i2,...,R(i1,i2,...,ik,...,in),...,in)
```

then for all $i1, i2, ..., (omitik), ..., in$, the vector B($i1, i2, ..., :, ..., in$) is sorted in ascending order; moreover, R($i1, i2, ..., :, ..., in$) 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 B($i1, i2, ..., j, ..., in$) = B($i1, i2, ..., m, ..., in$), then R($i1, i2, ..., j, ..., in$) $\leq$ R($i1, i2, ..., m, ..., in$).

If the optional DIM argument is absent, then the result S is an array of rank two, with shape (/ SIZE(SHAPE(ARRAY)), PRODUCT(SHAPE(ARRAY)) /) and the property that if one computes the rank-one array

```
B(k)=ARRAY(S(1,k),S(2,k),...,S(n,k))
```

where n=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 $j \leq m$ 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.

```
GRADE_DOWN(ARRAY,DIM)
```

The argument ARRAY may be of type integer, real, or character.

The result is an integer array.

If the optional DIM argument is present, then the result has the same shape as the ARRAY. Suppose DIM has the value k; then the result R has the property that if one computes the array

```
B(i1,i2,...,ik,...,in)=ARRAY(i1,i2,...,R(i1,i2,...,ik,...,in),...,in)
```

then for all $i1, i2, ..., (omitik), ..., in$, the vector B($i1, i2, ..., :, ..., in$) is sorted in descending order; moreover, R($i1, i2, ..., :, ..., in$) 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 B(i1,i2,...,j,...,in) .EQ. B(i1,i2,...,m,...,in), then R(i1,i2,...,j,...,in) $\leq$ R(i1,i2,...,m,...,in). (Note that the last "$\leq$" sign really should be a "$\leq$", not a "$\geq$".)

If the optional DIM argument is absent, then the result S is an array of rank two, with shape (/ SIZE(SHAPE(ARRAY)), PRODUCT(SHAPE(ARRAY)) /) and the property that if one computes the rank-one array

```
B(k)=ARRAY(S(1,k),S(2,k),...,S(n,k))
```

where n=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 ALL(S(:,j) .EQ. S(:,m)) will be false. The sort is stable; if j $\leq$ m and B(j) .EQ. 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.

Examples:

Because of the stability requirement, GRADE_DOWN(A(1:N)) does not, in general, equal GRADE_UP(A(N:1:-1)). Indeed, these results are equal if and only if A contains no duplicate values.

The stability requirement allows one to cascade grading operations in order to sort on multiple fields. For example, suppose one had the following derived type (example taken from section 4.4.1 of the Fortran 90 standard):

```
TYPE PERSON
  INTEGER AGE
  CHARACTER (LEN = 50) NAME
END TYPE PERSON
```

Now consider two arrays of persons:

```
TYPE(PERSON), DIMENSION(100000) :: MEMBERS, ROSTER
```

Also assume a work vector for indices:

```
INTEGER, DIMENSION(100000) :: V
```

Then the statements

```
V = GRADE_UP(MEMBERS%AGE, DIM=1)
V = V(GRADE_UP(MEMBERS(V)%NAME, DIM=1))
ROSTER = MEMBERS(V)
```

cause ROSTER to be a rearrangement of MEMBERS that is sorted primarily by name and secondarily by age (that is, members with the same name are grouped together in order of ascending age). Note that the minor sort field is graded first, and that more statements like the second one may be inserted to sort on additional fields. Without the use of the DIM argument, GRADE_UP returns a rank-two result of shape (/ 1, 100000 /), which would make the example more cumbersome.

To list members with the same name in descending order of age, change the first GRADE_UP to GRADE_DOWN:

```
V = GRADE_DOWN(MEMBERS%AGE, DIM=1)
V = V(GRADE_UP(MEMBERS(V)%NAME, DIM=1))
ROSTER = MEMBERS(V)
```

5.4.5 POPCNT, POPPAR, and LEADZ Functions

This section introduces three bit-manipulation functions.

POPCNT

An elemental, integer population count function. Its action on a scalar is:

```
POPCNT(x) = COUNT( (/ (BTEST(x,J), J=0, BIT_SIZE(x)-1) /) )
```

The result is the number of 1-bits in the integer x, according to the bit-manipulation model in section 13.5.7 of the Fortran 90 standard.

POPPAR

An elemental, integer population-parity function. Its action on a scalar is:

```
POPPAR(x) = MERGE(1,0,BTEST(POPCNT(x),0))
```

The result is 1 if the number of 1-bits in the integer x is odd, or 0 if the number of 1-bits in the integer x is even.

LEADZ

An elemental, integer count-leading-zeros function. Its action on a scalar is:

```
LEADZ(x) = MINVAL( (/ (J, J=0,BIT_SIZE(x)) /),    &
  MASK=(/ (BTEST(x,J), J=BIT_SIZE(x)-1,0,-1), .TRUE. /) )
```

The result is a count of the number of leading 0-bits in the integer x, according to the bit-manipulation model in section 13.5.7 of the Fortran 90 standard.

Note that a given integer value may produce different results from LEADZ, depending on the number of bits in the representation of the integer. That is because LEADZ counts bits from the most significant bit.

Chapter 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.

This chapter also 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.

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 is conceptually executing and but there may be 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. HPF refers 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 wr ng local code, because some facilities address the question of executing on multiple proc sors and local code by definition runs on a single processor.

The extrinsic procedure interface is an escape mechanism for calling a particular kind of non-HPF code from an HPF program, namely, code at is implemented as a local procedure on each processor. Such local procedures might resu rom an elaborate compilation process applied to a parallel programming language; they might also be identical copies of a SPMD procedure written in a conventional sequential language. HPF separates the question of local procedures as an implementation mechanism from the question of the programming language from which such local procedures are produced by a compiler.

A called procedure that is written in a language other than HPF and that uses the local procedure execution model should be declared **EXTRINSIC** within an HPF program that calls it.

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 sees a different picture. All HPF arrays accessible to the 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; the extrinsic procedure as a whole terminates only after every local procedure has terminated.

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 prescriptive data mapping directives, which are not relevant to local code; local procedures written in HPF may be intermixed with global HPF code. The use of HPF to define local procedures is discussed in Section 6.3.

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 this HPF specification to dictate implementation requirements for such languages or implementations. Nevertheless, a *recommended* way to use Fortran 90 to define local procedures is discussed in Section 6.4.

With the exception of return from a local procedure to the global caller that initiated local execution, there is no impl synchronization of the locally executing processors. A local procedure may use any co ol structure whatsoever. To access data outside the processor requires either preparatcry 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 proposal 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.)

This chapter is divided into three parts:

1. The HPF interface to extrinsic routines, and the contract between the caller and the callee.

2. A specific version of this interface for the case where extrinsic procedures are defined as explicit Single Program Multiple Data (SPMD) code with each local procedure coded in HPF. 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. 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.

6.2 Extrinsic Procedure Interface

6.2.1 Definition and Invocation of Extrinsic Procedures

An explicit interface must be provided for each extrinsic procedure entry in the scope where it is called, using an interface block. This interface defines the "HPF view" of the extrinsic procedure. The HPF directive EXTRINSIC occurs in the specification part for each extrinsic procedure entry.

extrinsic-directive **is** EXTRINSIC [*procedure-name*]

Examples:

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

      INTERFACE OPERATOR (+)
        FUNCTION LATKES(X, Y)   RESULT(Z)
```

```
!HPF$       EXTRINSIC
            REAL, ARRAY(:,:) :: X
            REAL, ARRAY(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 interface
             REAL X(:), RKNISH
          END RKNISH

          FUNCTION CKNISH(X)           !extrinsic interface
!HPF$        EXTRINSIC
             COMPLEX X(:), CKNISH
          END CKNISH
        END INTERFACE
```

In the last example, 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. (If both kinds of knishes are extrinsic, then **EXTRINSIC** directives go in their individual declarations, not in the containing declaration of generic knishes.)

Convention

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 illegal.

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.

Restrictions

1. 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 only disallows explicit alignment and distribution directives that imply that a scalar dummy argument is not replicated on all processors.
2. Each dummy argument must be nonsequential.
3. Extrinsic procedures may not be **RECURSIVE**.
4. Extrinsic procedures may not have alternate returns.
5. Extrinsic procedures may not be invoked, either directly or indirectly, in the body of a **FORALL** construct or in the body of an **INDEPENDENT** loop.
6. A dummy argument may not be a procedure name.
7. A dummy argument may not have the **POINTER** attribute.

6.2.2 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, mapping directives appearing in the interface are binding—the compiler must obey these directives in calling extrinsic procedures. (The reason for this rule is that data mapping is explicitly visible in local routines). Consequently, 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 6.2.4.

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 resumes.

2. The original distribution of arguments (and of the result of an extrinsic function) is restored, if necessary.

6.2.3 Requirements on the Callee

The callee must satisfy the constraints implied by the explicit interface to the procedure.

1. IN/OUT restrictions declared in the interface for the local subroutine must be obeyed.

2. Replicated variables, if updated, must be updated consistently. More precisely, if a variable accessible to a local subroutine 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. (This applies by default to all scalar arguments and to the result variable of a scalar-valued extrinsic function.)

3. No HPF variable is modified, unless it could be modified by an HPF procedure with the same explicit interface.

The call to an extrinsic procedure that fulfills these rules is semantically equivalent to the execution of a "global" HPF procedure, with the specified external interface.

6.2.4 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. We shall refer to an array actual argument passed by an HPF caller as 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 6.3.2. 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.

6.3 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 definition of a local procedure must include the HPF **LOCAL** directive in its scope.

local-directive **is** LOCAL

Any program unit may be declared to be **LOCAL**, possibly including the main program. (There are no globally mapped data objects if the main program is local). A **LOCAL** directive implies that the procedure is **EXTRINSIC**.

A local program unit may invoke only local program units, or internal procedures. Local program units can use all HPF constructs except for **DISTRIBUTE**, **REDISTRIBUTE**, **ALIGN**, **REALIGN**, and **INHERIT** directives. The distribution query intrinsics **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.

6.3.1 Argument Association

If a dummy argument in the HPF 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 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 the extrinsic HPF interface defines 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
   FUNCTION MATZOH(X, Y)   RESULT(Z)
      !HPF$ EXTRINSIC
      REAL, DIMENSION(:,:) :: X
      REAL, DIMENSION(SIZE(X,1)) :: Y, Z
      !HPF$ ALIGN WITH X(:,*)  :: Y(:)
      !HPF$ DISTRIBUTE (BLOCK, CYCLIC) X
   END FUNCTION
END INTERFACE
```

The corresponding local HPF (or Fortran 90) procedure is specified as follows.

```
   FUNCTION MATZOH(XX, YY)   RESULT(ZZ)
      !HPF LOCAL
      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)
```

L_INDEX An array of local coordinates of an element of the local array `ARRAY`.

G_INDEX The array of coordinates of the same element in the global array `G_ARRAY`.

`L_INDEX` has intent `IN` and `G_INDEX` has intent `OUT`.

7. `CALL GLOBAL_TO_LOCAL(ARRAY, G_INDEX, L_INDEX, LOCAL)`

   G_INDEX An array of coordinates of an element of the global array `G_ARRAY`.

   L_INDEX The array of local coordinates of the same element in the local array `ARRAY`.

   LOCAL A logical variable—set to `.TRUE.` if the local array has a copy of the global array element, `.FALSE.` otherwise.

   `G_INDEX` has intent `IN`, `L_INDEX` and `LOCAL` have itent `OUT`. Both `L_INDEX` and `LOCAL` are optional. The value returned by `L_INDEX` is undefined when `LOCAL` returns with value `.FALSE.`.

6.3.3 Restrictions

A local HPF routine that is invoked from global HPF code has to fulfill certain restrictions (these follow from the restrictions on the extrinsic interface):

1. A dummy argument may not be a procedure name.

2. A dummy argument may not have the POINTER attribute.

3. A dummy array argument must have assumed shape, even when it is explicit shape in the interface. Note that the shape of a dummy array argument differs from the shape of the corresponding actual argument, unless there is a single executing processor.

4. Explicit mapping directives for dummy arguments and function result variables may not appear in the declaration section of 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.

5. The attributes (type, kind, rank, optional, intent) of the dummy arguments must match the attributes of the corresponding dummy arguments in the explicit interface.

6. The local procedure may not be recursive.

7. The local procedure may not have alternate returns.

Local procedures may invoke other local HPF routines. Local blocks of global arrays may be passed as actual parameters to these routines. Local procedures may not invoke, either directly or indirectly, (global) HPF procedures. They may not access global HPF data other then data that is accessible, either directly or indirectly, via the actual parameters.

A local procedure may have several `ENTRY` points. The HPF program must contain a seperate extrinsic interface for each entry point that can be invoked from the HPF program.

6.4 Local Routines Written in Fortran 90

The suggested interface to local SPMD routines written is 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 implementations be extended to support the distribution query routines `GLOBAL_ALIGNMENT`, `GLOBAL_TEMPLATE`, and `GLOBAL_DISTRIBUTION` and the `PROCID` derived type as described in Section 6.3.2. (Because these routines cannot be used in initialization or specification expressions, it is possible to implement them as generic procedures. A module might contain the specification of these procedures and of the new `PROCID` type, so that no additions are required to Fortran 90 code beyond a `USE HPF` statement.)
- 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 restrictions listed in Section 6.3.3 ought to apply as well to local routines written in Fortran 90.

6.4.1 Argument Association

If a dummy argument in the HPF 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 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 the extrinsic HPF interface defines a function, then the local procedure is 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.

6.5 Example HPF Extrinsic Procedures

The first example shows an INTERFACE, 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
             SUBROUTINE  NEWMATMULT(A,B,C)
```

```
!HPF$       EXTRINSIC
            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.

      SUBROUTINE  NEWMATMULT(AA,BB,CC)
!HPF$ LOCAL
      REAL AA(:,:), BB(:,:), CC(:,:)
      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
      RETURN
      END
```

The second example shows an INTERFACE, 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.
```

```
         INTERFACE
               FUNCTION SREDUCE(A)
!HPF$          EXTRINSIC
               REAL, DIMENSION(NUMBER_OF_PROCESSORS()) :: SREDUCE
!HPF$ DISTRIBUTE (BLOCK) :: SREDUCE
               REAL,  DIMENSION(:), INTENT(IN)  ::  A
            END FUNCTION SREDUCE
         END INTERFACE
       ...
       TOTAL = SUM(SREDUCE(A))
       ...

! The Local Subroutine Definition
      SUBROUTINE  SREDUCE(AA)
!HPF$ LOCAL
      REAL AA(:)
      INTEGER COPIES

      CAll GLOBAL_ALIGNMENT(AA, NUMBER_OF_COPIES = COPIES)
      IF (COPIES > 1)
! ARRAY IS REPLICATED
     THEN CALL ERROR()

! ADDITIONAL CODE TO CHECK THAT TEMPLATE IS NOT REPLICATED
     ...
! ARRAY IS NOT REPLICATED -- COMPUTE LOCAL SUM
           ELSE RETURN(SUM(AA))
     END IF
     END
```

it likes, and distribute it as is appropriate for the machine, but it is not up to the user to say all this.

The proposals made to the IO subgroup were based on the following observations:

- A massively parallel machine needs massively parallel I/O;
- Efficient programs must avoid sequential bottlenecks from processors to file systems; and
- Fortran specifies that a file appears in element storage order; this conflicts with striped files (for example, an array distributed by rows may be written to a file striped by columns).

The proposals were that HPF should provide explicit control to obtain high performance I/O. In essence the three proposals were:

1. On a write, give a hint about how the data will be read.

   ```
   !HPF$ DISTRIBUTE (CYCLIC) :: a
   !HPF$ IO_DISTRIBUTE * :: a
   WRITE a, b, c
   ```

   When an array is written, it can be easily read back in the given distribution. The annotation can be associated with either the declaration or the write itself; in the first case it applies to all writes of the array, while in the second it only applies to the one statement. The intent is that metatdata is kept in the file system to record the "right" data layout. The advantages of this proposal include notation and efficiency

2. Give hints about the physical layout (number of spins, record length, striping function, etc.) of the file when it is opened.

   This uses the HPF array mapping mechanisms. (A file is a 1-dimensional array of records.) The syntax needs a "name" for the file "template"; the proposal is to use FILEMAP. The programmer can align/distribute FILEMAP (on I/O nodes), associate FILEMAP with a file on OPEN, etc. There are no changes in semantics or file system.

3. Introduce parallel read/write operations that are not necessarily compatible with sequential ones.

   ```
   PWRITE a
   PREAD a
   ```

   Data can be read back only into arrays of the same shape and mapping. Data written by PWRITE must be read by PREAD. This solution does not need metadata in file system or changes in the file system but is incompatible with the standard READ and WRITE.

Chapter 8

Sequence and Storage Association

High Performance Fortran (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.

8.1 Storage Association

8.1.1 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

8.1.4 Storage Association Rules

1. A *sequence-directive* with an empty *association-name-list* is treated as if it contained the name of all *association-names* in the scoping unit.

2. An aggregate cover may be explicitly mapped. If more than one aggregate cover exists for the aggregate variable group, only one may be explicitly mapped.

3. The only sequential variables that may be explicitly mapped are scalar or rank-one variables. Multi-dimensional sequential variables may not be explicitly mapped.

4. No explicit mapping may be given for an assumed-size dummy argument array.

5. No explicit mapping may be given for a component of a derived-data type having the Fortran 90 SEQUENCE attribute.

6. An HPF program is nonconforming if it specifies any mapping that would cause storage units to be mapped onto more than one abstract processor.

7. 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 and 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.

8.1.5 Storage Association Discussion

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.

As the examples below illustrate, there are many ways to use EQUIVALENCE with COMMON blocks that impact mappability of the variables in subtle ways. In order to allow maximum optimization for performance, the default for variables is to consider them mappable. In order to get correct separate compilation for scoping units that use COMMON blocks with different aggregate variable groups in different scoping units, it will be necessary to insert the HPF SEQUENCE directive. This is an example of where a correct FORTRAN 77 or Fortran 90 program will not necessarily be correct, without modification, in HPF.

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 a COMMON are the same in every scoping unit if the COMMON is not declared to be sequential. This same check should also verify that identical mappings have been selected for the variables in COMMON. Implementations without interprocedural information can use a link-time check. The second implementation option recommended is a global mechanism to declare that all 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.

Since an HPF program is nonconforming if it specifies any mapping that would cause storage units to be mapped onto more than one abstract processor, this puts 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 their individual numeric storage units are split because of some EQUIVALENCE statement or COMMON block layout, are nonconforming.

8.1.6 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. 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.

8.2 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 rank of a dummy argument to differ from the rank 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.

8.2.1 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 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 and the corresponding dummy argument differs from the actual argument in shape, then the dummy argument must be declared sequential and the actual argument must be an entire, sequential array.

3. A variable of type character (scalar or array) is nonsequential if it conforms to the requirements of Definition 5 of Section 8.1.1. 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.

8.2.2 Discussion of Sequence Association

Correct FORTRAN 77 (and hence Fortran 90) codes potentially require modification in HPF.. Explicit HPF SEQUENCE directives may be needed in some cases where sequence association exists in order for arrays to be mappable.

When the rank 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 8.1.1.

The ideal method for porting such codes will be to use an array section as the actual argument, which will allow both the dummy argument and its associated actual argument to be mappable.

Examples: Given

```
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

```
CALL HOME (ET)
```

requires either that ET and X are both sequential arrays or that ET is dimensioned exactly as X.

There is a special consideration for variables of type character. Change of the length of character variables across a call, as in

```
CHARACTER (LEN=100) one_long_word
CALL webster ( one_long_word )

SUBROUTINE webster(  short_dictionary  )
CHARACTER (LEN=5) short_dictionary ( 20 )
```

is conceptually legal in FORTRAN 77 and Fortran 90. It is allowed provided both actual argument and dummy argument are sequential. This may mean that !HPF$ SEQUENCE directives are needed.

Chapter 9

Subset High Performance Fortran

High Performance Fortran is defined as full Fortran 90 with restrictions on storage association and augmented by language features in three areas:

- Directives in the form of structured comments which, although they do not change the meaning of a program, provide information to a compiler to enable optimization;
- A `FORALL` statement and construct; and
- Extended intrinsic functions and a library.

This chapter presents a subset of HPF capable of being implemented more rapidly than the full HPF. A subset implementation will provide a standard interim capability and 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.

9.1 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 sequence and storage association.
- The Fortran 90 definitions of MIL-STD-1753 features:
  - `DO WHILE` statement (8.1.4.1.1 / R821)
  - `END DO` statement (8.1.4.1.1./ R825)
  - `IMPLICIT NONE` statement (5.3 / R540)
  - `INCLUDE` line (3.4)
  - scalar bit manipulation intrinsic procedures: `IOR`, `IAND`, `NOT`, `IEOR`, `ISHFT`, `ISHFTC`, `BTEST`, `IBSET`, `IBCLR`, `IBITS`, `MVBITS` (13.13)
  - binary, octal and hexadecimal constants for use in `DATA` statements (4.3.1.1 / R407 and 5.2.9 / R533)

- Arithmetic and logical array features:
  - array sections (6.2.2.3 / R618-621)
    - subscript triplet notation (6.2.2.3.1)
    - vector-valued subscripts (6.2.2.3.2)
  - array constructors limited to one level of implied DO (4.5 / R431)
  - arithmetic and logical operations on whole arrays and array sections (2.4.3, 2.4,5, and 7.1)
  - array assignment (2.4.5, 7.5, 7.5.1.4, and 7.5.1.5)
  - masked array assignment (7.5.3)
    - **WHERE** statement (7.5.3 / R738)
    - block **WHERE** . . . **ELSEWHERE** construct (7.5.3 / R739)
  - array-valued external functions (12.5.2.2)
  - automatic arrays (5.1.2.4.1)
  - **ALLOCATABLE** arrays and the **ALLOCATE** and **DEALLOCATE** statements (5.1.2.4.3, 6.3.1 / R622, and 6.3.3 / R631)
  - assumed-shape arrays (5.1.2.4.2 / R516)

- Intrinsic procedures:

  The list of intrinsic functions and subroutines below is a combination of routines which are entirely new to Fortran and 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 values of DIM which are initialization expressions and hence deliver a known shape at compile time 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)

9.2 Discussion of the Fortran 90 Subset Features

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.

9.3 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;
- Alignment subscripts more complicated than a multiple of the alignment dummy with a constant offset $(m * i + n)$;
- The PROCESSOR_VIEW directive;
- The PURE directive;
- The *forall-construct*;
- The HPF library and the HPF_LIB module;
- Values of the optional DIM arguments to the Fortran 90 MAXLOC and MINLOC intrinsic functions that are not initialization expressions; and
- The EXTRINSIC directive, LOCAL directive, definition of extrinsic procedures, and the Fortran 90 SPMD binding.

9.4 Discussion of the HPF Extension Subset

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. There is a further restriction on the form of expressions that can be used when aligning one array to another, with only very simple expression forms allowed in the subset.

Only the simplest version of FORALL statement is required in the subset. Note that the omission of the PURE directive 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_LIB module is also not part of the subset.

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 move between different HPF subset implementations.

Appendix A

Journal of Development

A.1 Nested WHERE statements

Briefly put, the less WHERE is like IF, the more difficult it is to translate existing serial codes into array notation. Such codes tend to have the general structure of one or more DO loops iterating over array indices and surrounding a body of code to be applied to array elements. Conversion to array notation frequently involves simply deleting the DO loops and changing array element references to array sections or whole array references. If the loop body contains logical IF statements, these are easily converted to WHERE statements. The same is true for translating IF-THEN constructs to WHERE constructs, except in two cases. If the IF constructs are nested (or contain IF statements), or if ELSE IF is used, then conversion suddenly becomes disproportionately complex, requiring the user to create temporary variables or duplicate mask expressions and to use explicit .AND. operators to simulate the effects of nesting.

Users also find it confusing that ELSEWHERE is syntactically and semantically analogous to ELSE rather than to ELSE IF.

We therefore propose that the syntax of WHERE constructs be extended and changed to have the form

```
where-construct         is  where-construct-stmt
                                [ where-body-construct ]...
                                [ elsewhere-stmt
                                [ where-body-construct ]... ]...
                                [ where-else-stmt
                                [ where-body-construct ]... ]
                                end-where-stmt

where-construct-stmt    is  WHERE ( mask-expr )

elsewhere-stmt          is  ELSE WHERE ( mask-expr )

where-else-stmt         is  ELSE WHERE

end-where-stmt          is  END WHERE

mask-expr               is  logical-expr

where-body-construct    is  assignment-stmt
                        or  where-stmt
                        or  where-construct
```

Constraint: In each assignment-stmt, the mask-expr and the variable being defined must be arrays of the same shape. If a where-con uct contains a where-stmt, an elsewhere-stmt, or another where-construct, then the two sk-expr's must be arrays of the same shape.

The meaning of such statements may be understood by rewrite rules. First one may eliminate all occurrences of ELSE WHERE:

```
WHERE (m1)                          WHERE (m1)
  xxx                                 xxx
ELSE WHERE (m2)       becomes       ELSE
  yyy                                 WHERE (m2)
END WHERE                               yyy
                                      END WHERE
                                    END WHERE
```

where xxx and yyy represent any sequences of statements, so long as the original WHERE, ELSE WHERE, and END WHERE match, and the ELSE WHERE is the first ELSE WHERE of the construct (that is, yyy may include additional ELSE WHERE or ELSE statements of the construct). Next one eliminates ELSE:

```
WHERE (m)                           temp = m
  xxx                               WHERE (temp)
ELSE                   becomes        xxx
  yyy                               END WHERE
END WHERE                           WHERE (.NOT. temp)
                                      yyy
                                    END WHERE
```

Finally one eliminates nested WHERE constructs:

```
WHERE (m1)                          temp = m1
  xxx                               WHERE (temp)
  WHERE (m2)                          xxx
    yyy                becomes      END WHERE
  END WHERE                         WHERE (temp .AND. (m2))
  zzz                                 yyy
END WHERE                           END WHERE
                                      WHERE (temp)
                                        zzz
                                      END WHERE
```

and similarly for nested WHERE statements.

The effects of these rules will surely be a familiar or obvious possibility to all the members of the committee; I enumerate them explicitly here only so that there can be no doubt as to the meaning I intend to support.

Such rewriting rules are simple for a compiler to apply, or the code may easily be compiled even more directly. But such transformations are tedious for our users to make by hand and result in code that is unnecessarily clumsy and difficult to maintain.

One might propose to make WHERE and IF even more similar by making two other changes. First, require the noise word THERE to appear in a WHERE and ELSE WHERE statement after the parenthesized mask-expr, in exactly the same way that the noise word THEN must appear in IF and ELSE IF statements. (Read aloud, the results might sound a trifle old-fashioned–"Where knights dare not go, there be dragons!"–but technically would be as grammatically correct English as the results of reading an IF construct aloud.) Second, allow a WHERE construct to be named, and allow the name to appear in ELSE WHERE, ELSE, and END WHERE statements. I do not feel very strongly one way or the other about these no doubt obvious points, but offer them for your consideration lest the possibilities be overlooked.

Now, for compatibility with Fortran 90, HPF should continue to use ELSEWHERE instead of ELSE, but this causes no ambiguity:

```
WHERE(...)
  ...
ELSE WHERE(...)
  ...
ELSEWHERE
  ...
END WHERE
```

is perfectly unambiguous, even when blanks are not significant(fixed source form). Since X3J3 declined to adopt the keyword THERE, it should not be used in HPF either (alas), though it could be allowed optionally.

A.2 ALLOCATE in FORALL

Proposal: ALLOCATE, DEALLOCATE, and NULLIFY statements may appear in the body of a FORALL.

Rationale: These are just another kind of assignment. They may have a kind of side effect (storage management), but it is a benign side effect (even milder than random number generation).

Example:

```
    TYPE SCREEN
      INTEGER, POINTER :: P(:,:)
    END TYPE SCREEN
    TYPE(SCREEN) :: S(N)
    INTEGER IERR(N)
    ...
!  Lots of arrays with different aspect ratios
    FORALL (J=1:N)  ALLOCATE(S(J)%P(J,N/J),STAT=IERR(J))
    IF(ANY(IERR)) GO TO 99999
```

A.3 Generalized Data References

Proposal: Delete the constraint in section 6.1.2 of the Fortran 90 standard (page 63, lines 7 and 8):

Constraint: In a data-ref, there must not be more than one part-ref with nonzero rank. A part-name to the right of a part-ref with nonzero rank must not have the POINTER attribute.

Rationale: Further opportunities for parallelism.
Example:

```
TYPE MONARCH
    INTEGER, POINTER :: P
END TYPE MONARCH
TYPE(MONARCH) :: C(N), W(N)
        ...
! Munch that butterfly
C = C + W * A%P ! Illegal in Fortran 90
```

A.4 FORALL with INDEPENDENT Directives

We propose that two new directives be added for use within the FORALL construct.

```
!HPF$BEGIN INDEPENDENT
!HPF$END INDEPENDENT
```

The two directives must be used in pair. A sub-block of statements parenthesized in the two directives is called an *asynchronous* sub-block or *independent* sub-block. The statements that are not in an asynchronous sub-block are in *synchronized* sub-blocks or *non-independent* sub-block. The synchronized sub-block is the same as Guy Steele's synchronized FORALL statement, and the asynchronous sub-block is the same as the FORALL with the INDEPENDENT directive. Thus, the block FORALL

```
      FORALL (e)
        b1
!HPF$BEGIN INDEPENDENT
        b2
!HPF$END INDEPENDENT
        b3
      END FORALL
```

means the same as

```
      FORALL (e)
        b1
      END FORALL
!HPF$INDEPENDENT
      FORALL (e)
        b2
      END FORALL
      FORALL (e)
```

```
    b3
  END FORALL
```

The INDEPENDENT directives indicates to the compiler there is no dependence and consequently, synchronizations are not necessary. It is users' responsibility to ensure there is no dependence between instances in an asynchronous sub-block.

A.4.1 What Does "No Dependence Between Instances" Mean?

It means that there is no true dependence, anti-dependence, or output dependence between instances. Examples of these dependences are shown below:

1. True dependence:

   ```
   FORALL (i = 1:N)
     x(i) = ...
     ...  = x(i+1)
   END FORALL
   ```

   Notice that dependences in FORALL are different from that in a DO loop. If the above example was a DO loop, that would be an anti-dependence.

2. Anti-dependence:

   ```
   FORALL (i = 1:N)
     ...  = x(i+1)
     x(i) = ...
   END FORALL
   ```

3. Output dependence:

   ```
   FORALL (i = 1:N)
     x(i+1) = ...
     x(i) = ...
   END FORALL
   ```

A.4.2 Rationale

1. A FORALL with a single asynchronous sub-block is the same as a DO with an INDEPENDENT assertion. A FORALL no INDEPENDENT directive is the same as a tightly synchronized FORALL. We only need to define one type of parallel constructs including both synchronized and asynchronous blocks. Furthermore, combining asynchronous and synchronized FORALLs, we have a loosely synchronized FORALL which is more flexible for many loosely synchronous applications.

2. With INDEPENDENT directives, the user can indicate which block needs not to be synchronized. The INDEPENDENT directives can act as barrier synchronizations.

A.5 EXECUTE-ON-HOME and LOCAL-ACCESS Directives

The EXECUTE-ON-HOME directive is used to suggest where an iteration of a DO construct or an indexed parallel assignment should be executed. The specified location of computation provides the reference with which the compiler determines which data access of the computation should be local and which data access may be remote. The LOCAL-ACCESS directive further asserts which data accesses are indeed local.

execute-on-home-directive	is	`EXECUTE (`*align-source-list*`) ON_HOME` *align-spec* `[,` *local-access-directive*`]`
local-access-directive	is	`LOCAL_ACCESS` *array-name-list*

The EXECUTE-ON-HOME directive must immediately precede the corresponding DO construct, array assignment, FORALL statement, FORALL construct or individual assignment statement in a FORALL construct.

The scope of an EXECUTE-ON-HOME directive is the entire loop body of the following DO construct, or the following array assignment, FORALL statement, FORALL construct or assignment statement in a FORALL construct.

When an EXECUTE-ON-HOME directive is applied to a DO construct, a FORALL statement, a FORALL construct or an assignment statement in a FORALL construct, the *align-source-list* identifies a distinct iteration index or an indexed parallel assignment in the corresponding scope and the *align-spec* identifies a template node. Every iteration index or indexed assignment must be associated with one and only one template node. The EXECUTE-ON-HOME directive states that each iteration or indexed parallel assignment should be executed on the processor to where its associated template node is mapped. For any subroutine call within a DO construct, the EXECUTE-ON-HOME directive specifies only the execution location of the caller but not necessarily the execution location of the called subroutine.

When an EXECUTE-ON-HOME directive is applied to an array assignment statement, each *align-source* identifies positions spread along one dimension (:) or a collapsed dimension (*) of the assigned array, and the *align-spec* identifies the associated template or template section. (Replication, i.e. "*", is not allowed in *align-spec*.) The *align-source-list* must have the same rank as the assigned array. The associated template or template section must have the same size as the assigned array in all uncollapsed dimensions. The EXECUTE-ON-HOME directive states that, for each element in the assigned array, the corresponding evaluation and assignment should be executed on the processor to where the corresponding template element of the associated template is mapped. For example,

```
!HPF$ EXECUTE (:,*) ON_HOME T(2:N)
      A(1:N-1,2:N) = B(2:N,1:N-1)
```

A(1,j) = B(2,j-1) is executed on the processor to which T(2) is mapped, where j = 2,..,N.

(*Align-spec* in current HPF is restricted to simple expressions of *align-dummy*. Thus only regular data mapping is supported. If array elements or functions are allowed in *align-spec* for specifying irregular data mapping, the above EXECUTE-ON-HOME directive can also be used to address the corresponding computation location problem.)

EXECUTE-ON-HOME directives can be nested, but only the immediately preceding EXECUTE-ON-HOME directive is effective.

The optional LOCAL-ACCESS directive asserts that all data accesses to the specified *array-name-list* within the scope of the EXECUTE-ON-HOME directive can be handled as local data accesses if the related HPF data mapping directives are honored.

The LOCAL-ACCESS directive can also be used separately from the EXECUTE-ON-HOME directive. When used alone, it applies only to the immediately following statement or construct, and asserts that all specified data accesses are local data accesses provided that the immediately preceding EXECUTE-ON-HOME directive and all related HPF data mapping directives are honored. The assertion overrides any local-access assertions by the preceding EXECUTE-ON-HOME directive. It is an error when a LOCAL-ACCESS directive is not applied inside the scope of some EXECUTE-ON-HOME directive.

INDEPENDENT and EXECUTE-ON-HOME directives can be combined into a single HPF directive when they are applied to the same DO or FORALL construct,

combined-assert-directive	**is**	*assertion-directive-list*
assertion-directive	**is**	*independent-directive*
	or	*execute-on-home-directive*

Example 1

```
      REAL A(N), B(N), C(N)
!HPF$ TEMPLATE T(N)
!HPF$ ALIGN WITH T:: A, B, C
!HPF$ DISTRIBUTE T(CYCLIC(2))

!HPF$ INDEPENDENT, EXECUTE (I) ON_HOME T(2*I), LOCAL_ACCESS A, B, C
      DO I = 1, N/2
      ! we know that P(2*I-1) and P(2*I) is a permutation
      ! of 2*I-1 and 2*I
        A(P(2*I - 1)) = B(2*I - 1) + C(2*I - 1)
        A(P(2*I)) = B(2*I) + C(2*I)
      END DO
```

Example 2

```
      REAL A(N,N), B(N,N)
!HPF$ TEMPLATE T(N,N)
!HPF$ ALIGN WITH T:: A, B
!HPF$ EXECUTE (I,J) ON_HOME T(I+1,J-1)
      FORALL (I=1:N-1, J=2:N)   A(I,J) = A(I+1,J-1) + B(I+1,J-1)
```

Example 3

```
      REAL A(N,N), B(N.N)
!HPF$ TEMPLATE T(N,N)
!HPF$ ALIGN WITH T:: A, B

!HPF$ EXECUTE (:,:) ON_HOME T(2:N,1:N-1)
      A(1:N-1,2:N) = A(2:N,1:N-1) + B(2:N,1:N-1)
```

Example 4 The original program for this example is due to Michael Wolfe of Oregon Graduate Institute.

This program performs matrix multiplication $C = A \times B$ by a systolic algorithm. Note that without the EXECUTE-ON-HOME and LOCAL_ACCESS directive, the compiler will have a hard time detecting that all A, B and C accesses are actually local.

```
      REAL A(N,N), B(N,N), C(N,N)

      PARAMETER(NOP = NUMBER_OF_PROCESSORS())
!HPF$ REALIGNABLE B
!HPF$ TEMPLATE T(2*N,N)                ! to allow wrap around mapping
!HPF$ ALIGN (I,J) WITH T(I,J):: A, C
!HPF$ ALIGN B(I,J) WITH T(N+I,J)
!HPF$ DISTRIBUTE T(CYCLIC(N/NOP),*)    ! distributed by row blocks

      IB = N/NOP

      DO IT = 0, NOP-1

      ! rotate B by row-blocks
!HPF$ REALIGN B(I,J) WITH T(N-IT*IB+I,J)

      ! data parallel loop
!HPF$ INDEPENDENT
!HPF$ EXECUTE (IP) ON_HOME T(IP*IB+1,1), LOCAL_ACCESS A, B, C

        DO IP = 0, NOP-1
          ITP = MOD( IT+IP, NOP )
          DO I = 1, IB
            DO J = 1, N
              DO K = 1, IB
                C(IP*IB+I,J) = C(IP*IB+I,J) +
     1                         A(IP*IB+I,ITP*IB+K)*B(ITP*IB+K,J)
              ENDDO  ! K
            ENDDO  ! J
          ENDDO  ! I
        ENDDO  ! IP
        ENDDO  ! IT
```

A.6 Elemental Reference of Pure Procedures

Fortran 90 introduces the concept of "elemental procedures", which are defined for scalar arguments but may also be applied to conforming array-valued arguments. The latter type of reference to an elemental procedure is called an "elemental" reference. Examples are the mathematical intrinsics, e.g. SIN and the intrinsic subroutine MVBITS. However, Fortran 90 restricts elemental reference to a subset of the intrinsic procedures — programmers cannot define their own elemental procedures. We propose that pure procedures may also be referenced elementally, subject to certain additional constraints given below.

A.6.1 Elemental Reference of Pure Functions

A user-defined pure function may be referenced *elementally*, provided it satisfies the additional constraints that:

1. Its non-procedure dummy arguments and dummy result are scalar and do not have the POINTER attribute.
2. The length of any character dummy argument or result is independent of argument values (though it may be assumed, or depend on the lengths of other character arguments and/or a character result).

We call non-intrinsic pure functions that satisfy these constraints "elemental non-intrinsic functions".

The interpretation of an elemental reference of such a function is as follows (adapted from Section 12.4.3 of the Fortran 90 standard):

> A reference to an elemental non-intrinsic function or to an elemental intrinsic function is an elemental reference if one or more non-procedure actual arguments are arrays and all array arguments have the same shape. If any actual argument is a function, its result must have the same shape as that of the corresponding function dummy procedure.
>
> The result of such a reference has the same shape as the array arguments, and the value of each element of the result, if any, is obtained by evaluating the function using the scalar and procedure arguments and the corresponding elements of the array arguments. The elements of the result may be evaluated in any order.
>
> Example:

```
      INTERFACE
        REAL FUNCTION foo (x, y, z, dummy_func)
          !HPF$ PURE foo
          REAL, INTENT(IN) :: x, y, z
          INTERFACE        ! interface for 'dummy_func''
            REAL FUNCTION dummy_func (x)
              !HPF$ PURE dummy_func
        REAL, INTENT(IN) :: x
      END FUNCTION dummy_func
    END INTERFACE
  END FUNCTION foo
END INTERFACE

REAL a(100), b(100), c(100)

c = foo (a, 0.0, b, sin)
```

A.6.2 Elemental Reference of Pure Subroutines

A user-defined pure subroutine may be referenced *elementally*, provided it satisfies the additional constraints that:

1. Its non-procedure dummy arguments are scalar and do not have the POINTER attribute.
2. The length of any character dummy argument is independent of argument values (though it may be assumed, or depend on the lengths of other character arguments).

We call non-intrinsic pure subroutines that satisfy these constraints "elemental non-intrinsic subroutines".

The interpretation of an elemental reference of such a subroutine is as follows (adapted from Section 12.4.5 of the Fortran 90 standard):

> A reference to an elemental non-intrinsic subroutine or an elemental intrinsic subroutine is an elemental reference if all actual arguments corresponding to INTENT(OUT) and INTENT(INOUT) dummy arguments are arrays that have the same shape and the remaining non-procedure actual arguments are conformable with them. If any actual argument is a function, its result must have the same shape as that of the corresponding function dummy procedure.
>
> The values of the elements of the arrays that correspond to INTENT(OUT) and INTENT(INOUT) dummy arguments are the same as if the subroutine were invoked separately, in any order, using the scalar and procedure arguments and corresponding elements of the array arguments.
>
> Example:

```
INTERFACE
  SUBROUTINE solve_simul(tol, y, z)
    !HPF$ PURE solve_simul
    REAL, INTENT(IN) :: tol
    REAL, INTENT(INOUT) :: y, z
  END SUBROUTINE
END INTERFACE

REAL a(100), b(100), c(100)
```

```
    CALL solve_simul( 0.1, a, b )
    CALL solve_simul( c(10:100:10), a(1:10), b(1:10) )
```

A.6.3 Constraints

It is perhaps worth outlining the reasons for the extra constraints imposed on pure procedures in order for them to be referenced elementally.

The dummy result of an elemental function or "output" arguments of a subroutine are not allowed to have the POINTER attribute because Fortran 90 does not permit an array of pointers to be referenced. The "input" arguments of an elemental reference are prohibited from having the POINTER attribute for consistency with the output arguments or result.

In an elemental reference, any actual argument that is a function must have a result whose shape agrees with that of the corresponding function dummy procedure. That is, elemental usage does not extend to function arguments, as Fortran 90 does not support the concept of an "array" of functions.

Finally, the length of any character dummy argument or a character dummy result cannot depend on argument *values* (though it can be assumed, or depend on the lengths of other character arguments). This ensures that under elemental reference, all elements of an array argument or result of character type will have the same length, as required by Fortran 90.

A.7 Parallel I/O

High Performance Fortran is primarily designed to obtain high performance on massively parallel computers. Such massively parallel machines also need massively parallel I/O.

There are difficulties in getting high performance I/O:

- Efficient programs must avoid sequential bottlenecks from processors to file systems
- Fortran specifies that a file appears in element storage order; this conflicts with striped files (for example, an array distributed by rows may be written to a file striped by columns).

In particular Fortran file organization has limits:

- Files have a sequential organization. (Even direct access files have records in sequential order, though they can be accessed out of order)
- Fortran files are record oriented
- Storage and sequence association are in force (when writing and then reading a file, for instance)
- No specification of the physical organization is possible
- No compatibility with other languages/machines is guaranteed

With these in mind there are two major approaches that have been suggested:

1. Define hints (annotations) that do not change file semantics, in the spirit of data distribution. (This gives some information to the compiler.)

2. Introduce parallel read/write operations that are not necessarily compatible with sequential ones.

A.7.1 Hints

Two ideas have been advanced which use the idea of giving hints to the compiler without changing the Fortran file semantics.

The first is based on the observation that although the distribution of an array when it is written may be available to the compiler or runtime system, the distribution into which that array will be read cannot generally be known, even though the programmer may have this knowledge. So the proposal is to provide on a write a hint about how the data will be read.

```
DISTRIBUTE (CYCLIC) :: a
!HPF$ IO_DISTRIBUTE * :: a
WRITE a, b, c
!HPF$ IO_DISTRIBUTE * :: b
```

When an array is written, it can be easily read back in the given distribution. The annotation can be associated with either the declaration or the write itself; in the first case it applies to all writes of the array, while in the second it only applies to the one statement. The intent is that meta-data is kept in the file system to record the "right" data layout. The advantages of this proposal include notation and efficiency

The second proposal is to give hints about the physical layout (number of spins, record length, striping function, etc.) of the file when it is opened.

This uses the HPF array mapping mechanisms. (A file is a 1-dimensional array of records.) The syntax needs a "name" for the file "template": we suggest FILEMAP. The programmer can align/distribute FILEMAP (on I/O nodes), associate FILEMAP with a file on OPEN, etc. There are again no changes in semantics or file system.

Mapping Files A Fortran file is a sequence of records. We treat such file as a 1-D array of records with LB=1 and infinite UB. This array can be mapped to a (storage) node arrangement in a manner analogous to the mapping of an array to a (processor) node arrangement. Files are mapped using the same notation as for array mapping. The mapping defines a partition of the file, and each part is associated with one abstract node.

The mapping of a file to a node arrangement can be interpreted in two ways:

1. The nodes may represent (abstract) independent storage units, each storing a fixed part of the file.

2. The nodes may represent (abstract) independent file caches, with a fixed association of each cache with a part of the file.

In both cases the file is mapped onto physical I/O devices so as to allow maximal concurrency for accesses directed to distinct parts of the file. If the second interpretation is used, then it is meaningful to align arrays and files onto the same templates.

We introduce a new filemap object. Filemaps are, essentially, named files. They appear where an array names would appear in a array mapping expression. An actual file is associated with a FILEMAP in an OPEN statement. Filemaps are introduced because files are not first class objects in FORTRAN (files are not declared). Also, Filemaps can have rank > 1, giving more flexibility in the types of mappings that can be specified.

The following diagram illustrates the mapping

```
                                          Node          Physical
File      Filemap      Template     arrangement      storage units
                                                      (or caches)

 _          _            _            _                 _
|_|-------->|_|-------->|_|-------->|_|------------>|_|

     OPEN         ALIGN      DISTRIBUTE    Implementation
                                              Dependent
```

Node Directive We suggest to replace the keyword `PROCESSOR` with the keyword `NODE`, which is more neutral. Node arrangements (ex processor arrangements) can be targets both for file mappings and for array mappings. Some implementations may disallow the use of the same node arrangement name as a target both for array mappings and for file mappings. In such case an `AFFINITY` directive, that specifies affinity between io nodes and processor nodes, would be useful. (Such directive would also be useful to specify affinity between nodes of different arrangements, e.g. nodes in arrangements of different rank.)

The set of allowable node arrangements that can be used to map files is implementation dependent – however, a node arrangement with `NUMBER_OF_IONODES` nodes is always legal.

The mapping of nodes to physical storage units is implementation dependent.

For example:

```
!HPF$ NODE :: D1(2,4), D2(2,2)
      PARAMETER(NOD=NUMBER_OF_IONODES())
!HPF$ NODE, DIMENSION(NOD) :: D3,D4
```

FILEMAP Directive A Fortran file is an infinite one-dimensional array of records, with LB=1. A filemap can be thought of as an assumed-size array of records. This array is associated with (one-dimensional) files, using storage association rules. The filemap name is used to specify a mappings for files. The association between a filemap name and an actual file is effected by the OPEN statement.

A FILEMAP directive declares filemap names. The syntax is

filemap-directive **is** `FILEMAP [::]`
filemap-name (*assumed-size-spec*)
[, *filemap-name* (*assumed-size-spec*)] ...
or `FILEMAP, DIMENSION (` *assumed-size-spec*)
:: *filemap-name-list*

An *assumed-size-spec* is a specification of the form used for assumed sized arrays: All dimensions are specified, with the exception of the last, which is assumed. In our case, the last dimension is infinite. Only initialization expressions may occur in this specification (including expressions that depend on `NUMBER_OF_IONODES`).

For example:

```
!HPF$ FILEMAP :: F1(2,4,*)
!HPF$ FILEMAP, DIMENSION(2,2,1:*) :: F2,F3
```

A FILEMAP directive does not allocate space, neither in memory, nor on disk.

File mapping ALIGN and DISTRIBUTE statements are used to map FILEMAPs onto nodes. The syntax is identical to the syntax for processor mappings, with one restriction: Block distributions cannot be used for the last (infinite) dimension of the filemap.

For example:

```
!HPF$ DISTRIBUTE (CYCLIC,CYCLIC,*) ONTO D2 :: F2,F3
!HPF$ DISTRIBUTE F1(*,BLOCK,CYCLIC(2)) ONTO D1
```

Assume that F1, F2 are the filemaps and D1, D2 are the node arrangements from the previous examples.

The first distribute statement specifies the following mapping for successive records of a file associated with F2 or F3.

```
D2(1,1)        D2(1,2)

1 (1,1,1)      3 (1,2,1)
5 (1,1,2)      7 (1,2,2)
.              .
.              .
.              .



D2(2,1)         D2(2,2)

2 (2,1,1)       4 (2,2,1)
6 (2,1,2)       8 (2,2,2)
.               .
.               .
```

The second distribute statement specifies the following mapping for successive records of a file associated with F1.

```
D1(1,1)         D1(1,2)        D1(1,3)         D1(1,4)

 1 (1,1,1)      17             33              49
 2 (2,1,1)      .              .               .
 3 (1,2,1)      .              .               .
 4 (2,2,1)      20             36              52
 9 (1,1,2)      25             41              57
10 (2,1,2)      .              .               .
```

```
11 (1,2,2)                              .                    .
12 (2,2,2)       28                     44                   60
65               81                     97                  113
 .                .                      .                    .
 .                .                      .                    .



D1(2,1)               D1(2,2)          D1(2,3)              D1(2,4)

 5 (1,3,1)            21               37                   53
 6 (2,3,1)             .                .                    .
 7 (1,4,1)             .                .                    .
 8 (2,4,1)            24               40                   56
13 (1,3,2)            29               45                   61
14 (2,3,2)             .                .                    .
15 (1,4,2)             .                .                    .
16 (2,4,2)            32               48                   64
69                    85              101                  117
 .                     .                .                    .
 .                     .                .                    .
```

OPEN statement

A new connection specifier of the form `FILEMAP = filemap-name` associates a mapping with the opened file. If the file exists then the mapping must be one of the mappings allowed for the file. The set of allowed file mappings for an existing file is implementation dependent, but always include the mapping under which the file was created. More generally, it will include any mapping where the file is mapped onto the same storage node arrangement, and with the same allocations of file records to storage nodes (different mappings may result in the same allocation of records to storage nodes). One choice is to allow any mapping, with possible degraded performance for ill matched mappings; another choice is to remap an existing file when it is opened with a new mapping, either offline or online. Vendors are expected to provide implementation dependent mechanisms to exercise such choices.

The default mapping is implementation dependent.

Only external files can be mapped.

Implementations may restrict the use of the FILEMAP connection specifier to files that are open for direct access (i.e., fixed size record files).

Parallel Data Transfer

The READ, WRITE, CLOSE, INQUIRE, BACKSPACE, ENDFILE, REWIND statements can be used to access distributed files; there are no changes in the syntax or semantics of these statements.

PREAD and PWRITE statements are added to allow efficient input or output of distributed arrays. The PREAD and PWRITE statements have the same syntax as unformatted I/O statements with READ or WRITE, respectively; they are semantically different.

The data representation created on a file by a PWRITE statement may be different from the data representation that obtains if PWRITE is replaced by WRITE. In particular, whereas an unformatted WRITE statement will create a single record (stored on one I/O node), a PWRITE statement may create multiple records, possibly on multiple I/O nodes. Whereas an unformatted READ statement accesses a unique record, a PREAD statement may access multiple records.

If a PWRITE statement was used to write a list of output items on a file, then a PREAD that starts at the same point in the file, and has a compatible list of input items, will return the values that were written. Two lists of items are compatible if the corresponding items in each list occupy the same number of storage units and have compatible mappings (informally, if the distribution of entries onto abstract processors is the same).

Examples

The program below exchanges the values of arrays A and B. The exchange is legal because the arrays are compatible.

```
REAL, DIMENSION(1000,1000) :: A, B
ALIGN A WITH B
...
OPEN(UNIT = 15, ACTION = READWRITE)
PWRITE (UNIT = 15) A, B
REWIND (UNIT = 15)
PREAD (UNIT = 15) B, A
```

The behavior of the program below is undefined. More than one record could have been created by the PWRITE statement, so that the BACKSPACE statement does not necessarily return the file position to where it was before PWRITE executed.

```
REAL, DIMENSION(1000,1000) :: A, B
ALIGN A WITH B ...
OPEN(UNIT = 15, ACTION = READWRITE)
PWRITE (UNIT = 15) A, B
BACKSPACE (UNIT = 15)
PREAD (UNIT = 15) B, A
```

The behavior of the program below is undefined, since the two arrays A and B don't have compatible distributions.

```
REAL, DIMENSION(1000,1000) :: A, B
DISTRIBUTE A(BLOCK,BLOCK)
DISTRIBUTE B(CYCLIC, CYCLIC)
...
OPEN(UNIT = 15, ACTION = READWRITE)
PWRITE (UNIT = 15) A, B
REWIND (UNIT = 15)
PREAD (UNIT = 15) B, A
```

Data written by a WRITE statement cannot be read with PREAD, and data written with PWRITE cannot be read with READ, or by a PREAD that does not start at exactly the same point in the file (otherwise the program outcome is undefined).

A.8 FORALL-ELSEFORALL construct

FORALL-ELSEFORALL construct is a natural generalization of Fortran 90 WHERE-ELSEWHERE construct. A construct proposed in previous drafts of the HPF:

```
FORALL(I=1:N,J=1:N)
  WHERE(MASK)
    assignment
  ELSEWHERE
    assignment
  ENDWHERE
ENDFORALL
```

seems to introduce unnecessary limitations coming from limitations of WHERE construct: the mask array must conform with the variables on the right side in all of the array assignment statements in the construct.

A.8.1 FORALL-ELSEFORALL Construct

The FORALL-ELSEFORALL construct is a generalization of the masked element array assignment statement allowing multiple assignments, masked array assignments, and nested FORALL statements to be controlled by a single *forall-triplet-spec-list.* Rule R215 for *executable-construct* is extended to include the *forall-construct.*

General Form of the FORALL-ELSEFORALL Construct

forall-construct	**is**	`FORALL` (*forall-triplet-spec-list* [, *scalar-mask-expr*]) *forall-body-stmt-list* [`ELSEFORALL`] [*elseforall-body-stmt-list*] `END FORALL`
forall-body-stmt	**is**	*forall-assignment*
	or	*forall-stmt*
	or	*forall-construct*
elseforall-body-stmt	**is**	*forall-body-stmt*

Constraint: *subscript-name* must be a *scalar-name* of type integer.
Constraint: A *subscript* or a *stride* in a *forall-triplet-spec* must not contain a reference to any *subscript-name* in the *forall-triplet-spec-list.*
Constraint: Any left-hand side *array-section* or *array-element* in any *forall-body-stmt* must reference all of the *forall-triplet-spec subscript-names.*
Constraint: If a *forall-stmt* or *forall-construct* is nested within a *forall-construct*, then the inner FORALL may not redefine any *subscript-name* used in the outer *forall-construct.* This rule applies recursively in the event of multiple nesting levels.

For each subscript name in the *forall-assignments*, the set of permitted values is determined on entry to the construct and is

$$m1 + (k-1) * m3, where\ k = 1, 2, ..., \lfloor \frac{m2 - m1 + 1)}{m3} \rfloor$$

PREAD and PWRITE can be used both for sequential access and for direct access. In the later case, the REC specifier indicates the position in the file where from the transfer starts. It is still the case that a transfer may involve several records.

Restrictions

The following restrictions allow for a simpler, more efficient implementation of parallel I/O. We may either put them in the language, or list them as recommended programming style.

1. Items in the item list of a PREAD or PWRITE statements are restricted to be variables (no io-implied-do). [Compilers may want to relax this rule, by considering an io-implied-do as being an operation that defines a new variable, akin to an array section, with a distribution induced by the distribution of the variables appearing in the implied-do-loop.]

2. All values needed to determine which entities are specified by a parallel I/O item list need be specified before the I/O statement. That is, we prohibit a statement of the form `PREAD (...)  N, A(1:N)`.

Extensions

- We may want to write an array with a layout that is suited to the mapping of the array that will appear in the input item list, rather than suited to the mapping of the array in the output list. To achieve this, we need to add align/distribute information as part of the PWRITE statement.

- We may want a `REMAP` statement, to be used instead of the sequence `CLOSE ... OPEN`, in order to associate a new mapping to an existing file.

- We may want to extend the `INQUIRE` statement to return file mapping information (alternatively, we may use the same query intrinsics used to query array partitions).

- A new intrinsics of the form `INDEX(filemap-name, list-of-indices)` would be handy, as it would allow to address random-access files as multi-dimensional arrays. E.g.

```
READ (7, REC = INDEX(F1,3,5) ) A
```

- Each data transfer operation specifies an association between parts of the file and abstract processor nodes where from (where to) the data in the record is transferred. We may want to add additional directives to the OPEN statement to indicate that this association fulfills certain restrictions for as long as the file is open.

  - Accesses to a file are *independent* if, in all data transfers, each file part is associated with the same processor node. An `INDEPENDENT` argument in the OPEN statement may be used to specify this condition (which simplifies file caching).
  - A data transfer is *aligned* if each file part is associated with a unique processor node (is not split among two processor nodes). We may use an `ALIGNED` argument in the OPEN statement to specify that all data transfers are aligned. (INDEPENDENT implies ALIGNED, but not vice versa).

and where *m1*, *m2*, and *m3* are the values of the first subscript, the second subscript, and the stride respectively in the *forall-triplet-spec*. If *stride* is missing, it is as if it were present with a value of the integer 1. The expression *stride* must not have the value 0. If for some subscript name $\lfloor (m2 - m1 + 1)/m3 \rfloor \leq 0$, the *forall-assignments* are not executed.

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 *subscript-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 *subscript-name* values. The mask elements may be evaluated in any order. One set of active combinations of *subscript-name* values is the subset of the valid combinations for which the mask evaluates to true and a second one is the subset of the valid combinations for which the mask evaluates to false.

3. Execute *forall-body-stmts* in the order they appear for the set of the valid combination of *subscript-name* for which mask was evaluated to true in the step 2. Each statement is executed completely (that is, for all active combinations of *subscript-name* values) according to the following interpretation:

   (a) Assignment statements, pointer assignment statements, and array assignment statements (i.e. statements in the *forall-assignment* category) evaluate the right-hand side *expr* and any left-and side subscripts for all active *subscript-name* values, then assign those results to the corresponding left-hand side references.

   (b) FORALL statements and FORALL constructs 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 *subscript-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 *subscript-names*. The scalar mask expression is then evaluated for all valid combinations of the inner FORALL's *subscript-names* to produce the set(s) of active combinations, as in step 2. If there is no scalar mask expression, it is assumed to be always true. Each statement in the inner FORALL is then executed for each valid combination (of the inner FORALL), recursively following the interpretations given in this section.

4. Execute *elseforall-body-stmts* for the set of active *subscript-name* for which the mask was evaluated to false in the step 2, the same way as in 3.

If the scalar mask expression is omitted, it is as if it were present with the value true. In that case ELSEFORALL statement is not allowed.

The scope of a *subscript-name* is the FORALL construct itself.

A single assignment or array assignment statement in a *forall-construct* must obey the same restrictions as a *forall-assignment* in a simple *forall-stmt*. (Note that the lowest level of nested statements must always be an assignment statement.) 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.

Scalarization of the FORALL-ELSEFORALL Construct

A *forall-construct* of the form:

```
FORALL ( v=l:u:s, mask )
    a(l:u:s) = rhs1
ELSEFORALL
    a(l:u:s) = rhs2
END FORALL
```

is equivalent to the following standard Fortran 90 code:

```
!evaluate subscript and stride expressions in any order

templ = l
tempu = u
temps = s

!then evaluate the FORALL mask expression

DO v=templ,tempu,temps
 tempmask(v) = mask
END DO

!then evaluate the masks

DO v1=templ,tempu,temps
    tempmask(v) = mask(v)
  END IF
END DO

!then evaluate the first block of statements

DO v=templ,tempu,temps
  IF (tempmask(v)) THEN
      temprhs1(v) = rhs1
  END IF
END DO
DO v1=templ,tempu,temps
  IF (tempmask(v)) THEN
   a(v)=temprhs1(v)
  END IF
END DO

!then evaluate the second block of statements

DO v=templ,tempu,temps
  IF (not.tempmask(v)) THEN
```

```
      temprhs2(v) = rhs2
    END IF
  END DO
  DO v1=templ,tempu,temps
    IF (.not.tempmask(v)) THEN
     a(v)=temprhs2(v)
    END IF
  END DO
```

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] P. Brezany, M. Gerndt, P. Mehrotra and H. Zima. *Concurrent File Operations in a High Performance Fortran*

[6] 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.

[7] 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.

[8] M. Chen and J. Cowie. *Prototyping Fortran 90 Compilers for Massively Parallel Machines*, SIGPLAN92, 1992.

[9] Digital Equipment Corporation, Maynard, Massachusetts. *DECmpp 12000 Sx - DECmpp Sx Parallel Fortran Reference Manual*, July 1992.

[10] 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.

[11] ISO. *Fortran 90*, May 1991. [ISO/IEC 1539: 1991 (E) and now ANSI X3.198-1992].

[12] David B. Loveman. *Element Array Assignment - the FORALL Statement*, Third Workshop on Compilers for Parallel Computers, Vienna, Austria, July 6-9, 1992.

[13] MasPar Computer Corporation, 749 North Mary Avenue, Sunnyvale, California. *MasPar Fortran Reference Manual*, May 1991. [Software Version 1.1, 9303-0000, Rev. A2].

[14] 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.

[15] Andrew Meltzer, Douglas M. Pase, and Tom MacDonald. *Basic Features of the MPP Fortran Programming Model*, Cray Research, Inc, Eagan, Minnesota, August 19, 1992.

[16] Michael Metcalf and John Reid. *Fortran 90 Explained*, Oxford University Press, 1990.

[17] Douglas M. Pase, Tom MacDonald, and Andrew Meltzer. *MPP Fortran Programming Model*, Cray Research, Inc, Eagan, Minnesota, August 26, 1992.

[18] Thinking Machines Corporation, Cambridge, Massachusetts. *CM Fortran Reference Manual*, July 1991.

[19] US Department of Defense. *Military Standard, MIL-STD-1753: FORTRAN, DoD Supplement to American National Standard X3.9-1978*, November 9, 1978.

[20] 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.