XcalableMP

⟨*ex-scalable-em-p*⟩

Language Specification

Version 1.3

XcalableMP Specification Working Group

November, 2016

History

Version 1.3: August, 2017

- 2.8.2 Node Terminology
- 2.8.6 Local-view Terminology
- 3.1 Array Section Notation
- 3.2 Array assignment statements in C
- 3.5 Dynamic Allocation of Global Data in C
- 4.2 `nodes` Directive
- 4.3.1 `template` Directive
- 4.3.4 `align` Directive
- 4.4.3 `loop` Construct
- 4.5.1 `reflect` Construct
- 4.5.6 `wait_async` Construct
- 4.5.8 `reduce_shadow` Construct
- 5.7 Coarrays in XcalableMP C
- 5.8 Directives for the Local-view Programming
- 7.2.2 `xmpc_all_node_num`
- 7.2.5 `xmpc_node_num`
- 7.2.6 `xmpc_this_image`
- 7.2.8 `xmp_num_images`
- 7.3 Execution Control Functions
- 7.5.1 `xmp_malloc`
- 7.6.4 `xmp_nodes_attr`
- B.2.1 `xmp_array_gtol`
- B.2.2 `xmp_array_lsize`
- B.2.4 `xmp_array_lda`
- 7.9.1 `xmp_scatter`
- 7.9.3 `xmp_pack`
- 7.9.4 `xmp_unpack`
- C Memory-layout Model

Version 1.2.1: November, 2014 Corrections and clarifications to Version 1.2.

- 4.3.4 `align` Directive
- 4.4.3 `loop` Directive
- 5.7.1 `[C]` Declarations of Coarrays
- 5.8.1 `[F] local_alias` Directive
- 3.1 Array Section Notation
- 3.2 Array Assignment Statement

Version 1.2: November 20, 2013

Version 1.1: November 13, 2012

Version 1.0: November 14, 2011

Contents

List of Figures

List of Tables

Acknowledgment

The XcalableMP specification is designed by the XcalableMP Specification Working Group, which consists of the following members from academia, research laboratories, and industries.

- Tatsuya Abe ..... RIKEN
- Tokuro Anzaki ..... Hitachi
- Taisuke Boku ..... University of Tsukuba
- Toshio Endo ..... TITECH
- Yoshinari Fukui ..... JAMSTEC
- Yasuharu Hayashi ..... NEC
- Atsushi Hori ..... RIKEN
- Kohichiro Hotta ..... Fujitsu
- Hidetoshi Iwashita ..... RIKEN
- Susumu Komae ..... AXE
- Atsushi Kubota ..... Hiroshima City University
- Jinpil Lee ..... University of Tsukuba
- Toshiyuki Maeda ..... RIKEN
- Motohiko Matsuda ..... RIKEN
- Yuichi Matsuo ..... JAXA
- Kazuo Minami ..... RIKEN
- Shoji Morita ..... AXE
- Hitoshi Murai ..... RIKEN
- Kengo Nakajima ..... University of Tokyo
- Takashi Nakamura ..... JAXA
- Tomotake Nakamura ..... RIKEN
- Mamoru Nakano ..... CRAY
- Masahiro Nakao ..... RIKEN
- Takeshi Nanri ..... Kyusyu University
- Kiyoshi Negishi ..... Hitachi
- Satoshi Ohshima ..... University of Tokyo
- Yasuo Okabe ..... Kyoto University
- Hitoshi Sakagami ..... NIFS
- Tomoko Sakari ..... Fujitsu
- Shoich Sakon ..... NEC
- Mitsuhisa Sato ..... University of Tsukuba
- Taizo Shimizu ..... PC Cluster Consortium
- Takenori Shimosaka ..... RIKEN
- Yoshihisa Shizawa ..... RIST

- Shozo Takeoka ..... AXE
- Hitoshi Uehara ..... JAMSTEC
- Eiji Yamanaka ..... Fujitsu
- Masahiro Yasugi ..... Kyushu Institute of Technology
- Mitsuo Yokokawa ..... Kobe University

This work was supported by "Seamless and Highly-productive Parallel Programming Environment for High-performance Computing" project funded by Ministry of Education, Culture, Sports, Science and Technology, Japan, and is supported by PC Cluster Consortium.

Chapter 1

Introduction

This document defines the XcalableMP specification, which is a directive-based language extension of Fortran and C for scalable and performance-aware parallel programming. The specification includes a collection of compiler directives and intrinsic and library procedures, and provides a model of parallel programming for distributed memory multiprocessor systems.

1.1 Features of XcalableMP

The features of XcalableMP are summarized as follows:

- XcalableMP supports typical parallelization based on the data-parallel paradigm and work mapping under the "global-view" programming model, and it enables the parallelization of the original sequential code using minimal modification with simple directives such as OpenMP [1]. Many ideas on "global-view" programming are inherited from High Performance Fortran (HPF) [2].

- The important design principle of XcalableMP is "performance-awareness." All actions related to communication and synchronization are taken by directives (and coarray features), which is different from automatic parallelizing compilers. The user should be aware of the effect of the XcalableMP directives in the execution model for distributed-memory architecture.

- XcalableMP also includes features from Partitioned Global Address Space (PGAS) languages, such as coarray of the Fortran 2008 standard, for "local-view" programming.

- An extension of existing base languages with directives is useful to reduce code-rewriting and education costs. The XcalableMP language specification is defined as an extension to the Fortran and C base languages.

- For flexibility and extensibility, the execution model enables us to combine XcalableMP with explicit Message Passing Interface (MPI) [3] coding for more complicated and tuned parallel codes and libraries.

- For multi-core and SMP clusters, OpenMP directives can be combined into XcalableMP for thread programming inside each node as a hybrid programming model.

XcalableMP is being designed based on experiences gained during the development of HPF, HPF/JA [4], Fujitsu XPF (VPP FORTRAN) [5, 6], and OpenMPD [7].

1.2 Scope

The XcalableMP specification covers only user-directed parallelization, where the user explicitly specifies the behavior of the compiler and the runtime system in order to execute the program in parallel in a distributed-memory system. XcalableMP-compliant implementations are not required to automatically distribute data, detect parallelism, parallelize loops, or generate communications and synchronizations.

1.3 Organization of this Document

The remainder of this document is structured as follows:

- Chapter 2: Overview of the XcalableMP Model and Language
- Chapter 3: Base Language Extensions in XcalableMP C
- Chapter 4: Directives
- Chapter 5: Support for Local-view Programming
- Chapter 6: Procedure Interface
- Chapter 7: Intrinsic and Library Procedures
- Chapter 8: OpenMP in XcalableMP Programs

In addition, the following appendices are included in this document as proposals.

- Appendix A: Programming Interface for MPI
- Appendix B: Interface to Numerical Libraries
- Appendix C: Memory-layout Model
- Appendix D: XcalableMP I/O

1.4 Changes to Version 1.3 from Version 1.2.1

- In XcalableMP C, a square bracket is available in *nodes-decl*, *nodes-ref*, *template-ref*, and *template-decl*.
- Add the `orthogonal` clause to the `reflect` directive in Section 4.5.1.
- Add `xmpc_all_node_num()` in Section 7.2.2.
- Add `xmpc_node_num()` in Section 7.2.5.
- Add `xmpc_this_image()` in Section 7.2.6.
- Add `xmp_num_images()` in Section 7.2.8.
- Modify `xmp_array_gtol()` in Section B.2.1.
- Change `xmp_array_lsize()` not to include shadow object in Section B.2.2.
- Create `xmp_array_lda()` from `xmp_array_lead_dim()` in Section B.2.4.

- In XcalableMP C, the dynamic allocataion of multi-dimensional global data is allowed.
- A restriction on the `align` directive is added.
- The `expand` and `margin` clauses of the `loop` construct are added.
- The meaning of a reduction-kind "`-`" in the `reduction` clause of the `loop` construct is changed.
- The treatment for *async-id* not associated with any asynchronous communication is specified.
- The `reduce_shadow` construct is added.
- The description of the `local_alias` directive is modified.
- The `xmp_exit` library function is added.
- The specifications of `xmp_scatter`, `xmp_pack`, and `xmp_unpack` are modified.
- The memory consistency model of XcalableMP is discussed in the appendix.

1.5 Changes to Version 1.1 from Version 1.2

- The position of `align` directives for dummy arguments in XcalableMP C is specified.
- It is specified that aligned arrays cannot be initialized.
- The interpretation of a `reduction` clause of the `loop` directive is corrected.
- The syntax for declaring coarrays is changed.
- An assumed-shape array can be the target of the `local_alias` directive.
- The syntax and the semantics of the array section notation in XcalableMP C is modified.
- The syntax of the array assignment statement in XcalableMP C is extended.

Chapter 2

Overview of the XcalableMP Model and Language

2.1 Hardware Model

The target of XcalableMP is distributed-memory multicomputers (Figure 2.1). Each computation node, which may contain several cores, has its own local memory (shared by the cores, if any), and is connected with the others via an interconnection network. Each node can access its local memory directly and remote memory (the memory of another node) indirectly (i.e., through inter-node communication). However, it is assumed that accessing remote memory is much slower than accessing local memory.

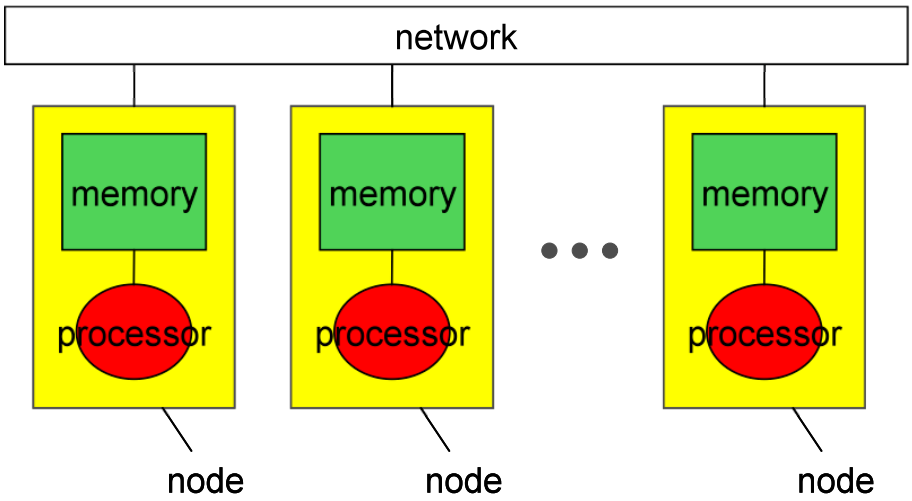

Figure 2.1: Hardware model.

2.2 Execution Model

An XcalableMP program execution is based on the Single Program Multiple Data (SPMD) model, where each node starts execution from the same main routine, and continues to execute the same code independently (i.e., asynchronously), which is referred to as the *replicated execution*, until it encounters an XcalableMP construct.

A set of nodes that executes a procedure, statement, loop, a block, etc. is referred to as its *executing node set*, and is determined by the innermost `task`, `loop`, or `array` directive surrounding it dynamically, or at runtime. The *current executing node set* is an executing node set of the current context, which is managed by the XcalableMP runtime system on each node.

The current executing node set at the beginning of the program execution, or *entire node set*, is a node set that contains all the available nodes, which can be specified in an implementation-defined way (e.g., through a command-line option).

When a node encounters at runtime either a `loop`, `array`, or `task` construct, and is contained by the node set specified by the `on` clause of the directive, it updates the current executing node set with the specified one and executes the body of the construct, after which it resumes the last executing node set and proceeds to execute the subsequent statements.

In particular, when a node in the current executing node set encounters a `loop` or an `array` construct, it executes the loop or the array assignment in parallel with other nodes, so that each iteration of the loop or element of the assignment is independently executed by the node in which a specified data element resides.

When a node encounters a synchronization or a communication directive, synchronization or communication occurs between it and other nodes. That is, such *global constructs* are performed collectively by the current executing nodes. Note that neither synchronization nor communication occurs unless these constructs are being specified.

2.3 Data Model

There are two classes of data in XcalableMP: *global data* and *local data*. Data declared in an XcalableMP program are local by default.

Global data are distributed onto the executing node set by the `align` directive (see section 4.3.4). Each fragment of distributed global data is allocated in the local memory of a node in the executing node set.

Local data comprises all data that are not global. They are replicated within the local memory of each of the executing nodes.

A node can access directly only local data and sections of global data that reside in its local memory. To access data in remote memory, explicit communication must be specified in ways such as global communication constructs and coarray assignments.

In particular, in XcalableMP Fortran, for common blocks that include any global variables, it is implementation-defined what storage sequences they occupy and how storage association is defined between two of them.

2.4 Global-view Programming Model

The global-view programming model is useful when, starting from a sequential version of a program, the programmer parallelizes it in data-parallel style by adding directives with minimum modification. In the global-view programming model, the programmer describes the distribution of data among nodes using the data distribution directives. The `loop` construct assigns each iteration of a loop to the node at which the computed data is located. The global-view communication directives are used to synchronize nodes, maintain the consistency of shadow areas, and move sections of distributed data globally. Note that the programmer must specify explicitly communications to make all data references in the program local, and this is done using appropriate directives.

In many cases, the XcalableMP program according to the global-view programming model is based on a sequential program, and it can produce the same results, regardless of the number

of nodes (Figure 2.2).

There are three groups of directives for the global-view programming model. Because these directives are ignored as a comment by the compilers of base languages (Fortran and C), an XcalableMP program can be compiled by them to ensure that they run properly.

Data Mapping

Specifies the data distribution and mapping to nodes (partially inherited from HPF).

Work Mapping (Parallelization)

Assigns a work to a node set. The `loop` construct maps each iteration of a loop to nodes owning a specific data elements. The `task` construct defines a set amount of work as a *task*, and assigns it to a specific node set.

Communication and Synchronization

Specifies how to communicate and synchronize with the other compute nodes. In XcalableMP, inter-node communication must be explicitly specified by the programmer. The compiler guarantees that no communication occurs unless it is explicitly specified by the programmer.

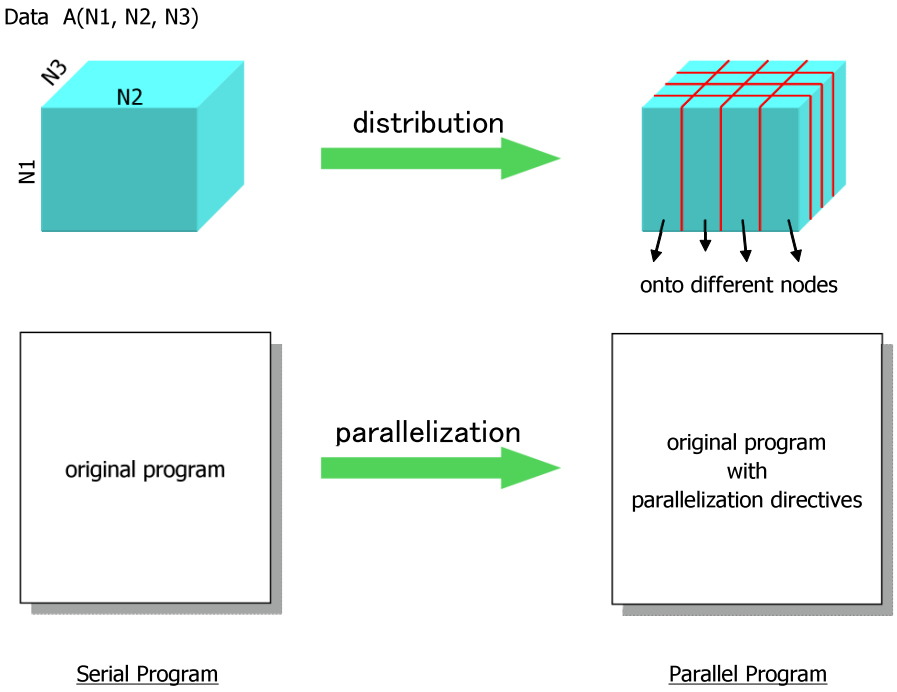

Figure 2.2: Parallelization using the global-view programming model.

2.5 Local-view Programming Model

The local-view programming model is suitable for programs that explicitly describe an algorithm and a remote data reference that are to be executed by each node (Figure 2.3).

For the local-view programming model, some language extensions and directives are provided. The coarray notation, which is imported from Fortran 2008, is one such extension, and can be

used to specify which replica of a local data is to be accessed. For example, the expression of `A(i)[N]` is used to access an array element of `A(i)` located on the node `N`. If the access is a reference, then a one-sided communication to get the value from the remote memory (i.e., the *get* operation) is issued by the executing node. If the access is a definition, then a one-sided communication to put a value to the remote memory (i.e., the *put* operation) is issued by the executing node.

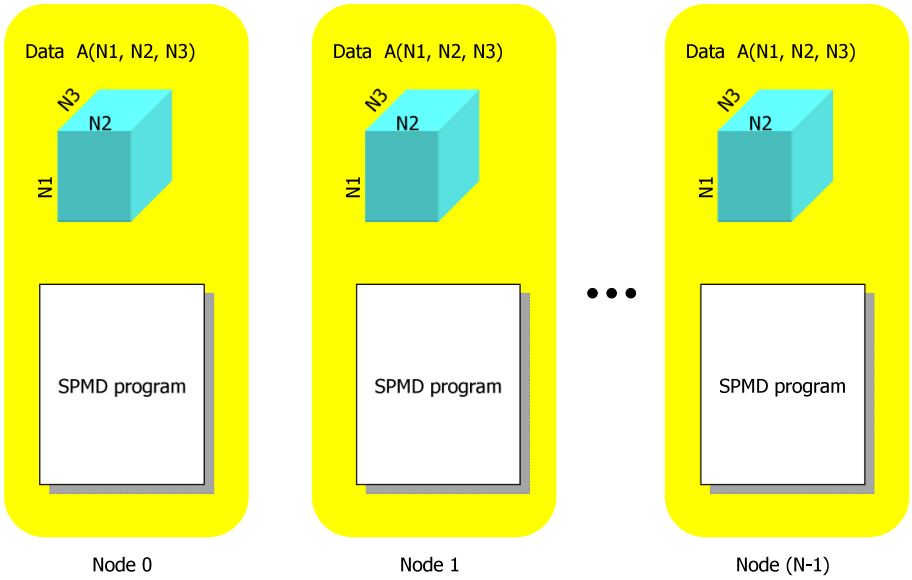

Figure 2.3: Local-view programming model.

2.6 Interactions between Global View and Local View

In the global view, nodes are used to distribute data and works. In the local view, nodes are used to address data in the coarray notation. In application programs, programmers should choose an appropriate data model according to the structure of the program. Figure 2.4 illustrates the global view and the local view of data.

Data may have both a global view and a local view, and can be accessed from either. XcalableMP provides some directives to give the local name (alias) to the global data declared in the global-view programming model to enable them to also be accessed in the local-view programming model. This feature is useful to optimize a certain part of the program by using explicit remote data access in the local-view programming model.

2.7 Base Languages

The XcalableMP language specification is defined based on Fortran and C as the base languages. More specifically, the base language of XcalableMP Fortran is Fortran 90 or later, and that of XcalableMP C is ISO C90 (ANSI C89) or later.

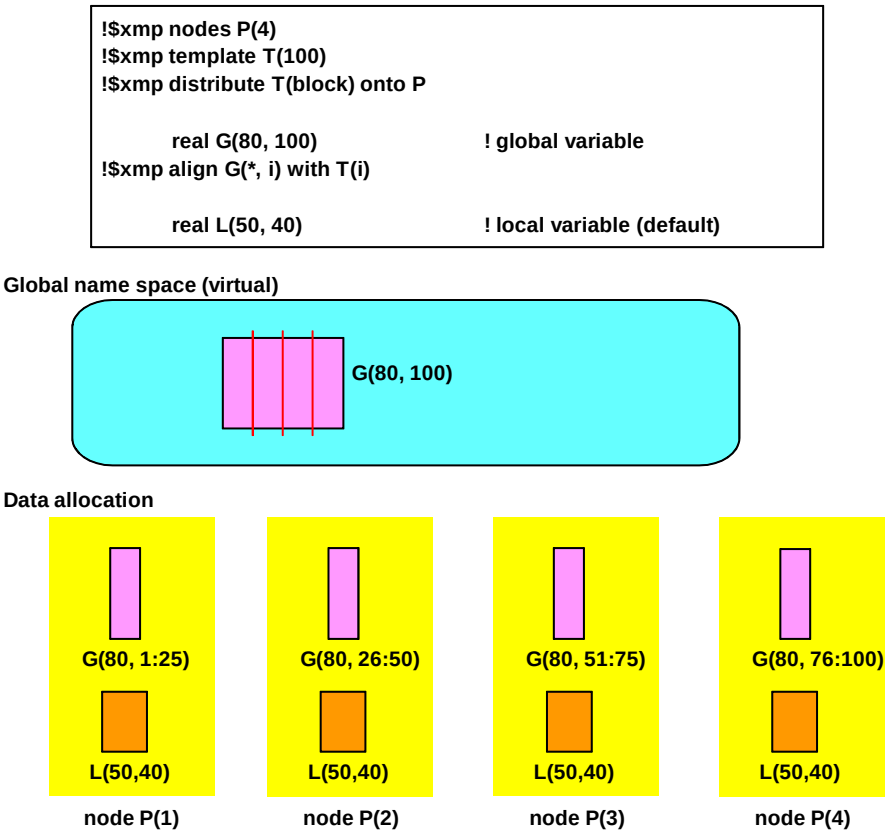

Figure 2.4: Global view and local view.

2.8 Glossary

2.8.1 Language Terminology

base language A programming language that serves as the foundation of the XcalableMP specification.

base program A program written in a base language.

XcalableMP Fortran The XcalableMP specification for a base language Fortran, abbreviated as XMP/F.

XcalableMP C The XcalableMP specification for a base language C, abbreviated as XMP/C.

structured block For C, an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom, or an XcalableMP construct. For Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom, or an XcalableMP construct.

procedure A generic term used to refer to "procedure" (including subroutine and function) in XcalableMP Fortran and "function" in XcalableMP C.

directive In XcalableMP Fortran, a comment, and in XcalableMP C, a `#pragma`, that specifies XcalableMP program behavior.

declarative directive An XcalableMP directive that may only be placed in a declarative context. A declarative directive has no associated executable user code; instead, it has one or more associated user declarations.

executable directive An XcalableMP directive that is not declarative; it may be placed in an executable context.

construct An XcalableMP executable directive (and for Fortran, the paired `end` directive, if any) and the associated statement, loop, or structured block, if any.

global construct A construct that is executed collectively and synchronously by every node in the current executing node set. Global constructs are further classified into two groups of *global communication constructs*, such as `gmove` and `barrier`, which specify communication or synchronization, and *work mapping constructs*, such as `loop`, `array`, and `tasks`, which specify parallelization of loops, array assignments, or tasks.

template A dummy array that represents an index space to be distributed onto a node set, which serves as the "template" of parallelization in XcalableMP and can be considered to represent a set of, for example, grid points in the grid method and particles in the particle method. A template is used in an XcalableMP program to specify the data and work mapping. Note that the lower bound of each dimension of a template is one in both XcalableMP Fortran and XcalableMP C.

data mapping Allocating elements of an array to nodes in a node set by specifying with the `align` directive that the array is aligned with a distributed template.

work mapping Assigning each of the iterations of a loop, the elements of an array assignment, or the tasks to nodes in a node set. Such work mapping is specified by aligning it with a template or distributing it onto a node set.

global A data or a work is *global* if and only if there are one or more replicated instances of it, each of which is shared by the executing nodes.

local A data or a work is *local* if and only if there is a replicated instance of it on each of the executing nodes.

global-view model A programming or parallelization model on which parallel programs are written by specifying how to map global data and works onto nodes.

local-view model A programming or parallelization model on which parallel programs are written by specifying how each node owns local data and performs local works.

2.8.2 Node Terminology

node An execution entity managed by the XcalableMP runtime system, which has its own memory and can communicate with other nodes. A node can execute one or more threads concurrently.

node set A totally ordered set of nodes.

entire node set A node set that contains all of the nodes participating in the execution of an XcalableMP program. It is the current executing node set specified explicitly or implicitly at the beginning of the program execution.

executing node set A node set that contains all of the nodes participating in the execution of a procedure, statement, construct, etc. of an XcalableMP program is called its executing node set. In this document, this term is used to represent the *current executing node set* unless it is ambiguous. The executing node set at the beginning of the program execution is the entire node set.

current executing node set An executing node set of the current context, which is managed by the XcalableMP runtime system. The current executing node set can be modified by the `task`, `array`, or `loop` constructs.

executing node A node in the executing node set.

node array An XcalableMP entity of the same form as a Fortran array that represents a node set in XcalableMP programs. Each element of a node array represents a node in the corresponding node set. A node array is declared by the `nodes` directive. Note that the lower bound of each dimension of a node array is one in both XcalableMP Fortran and XcalableMP C.

entire node array A node array corresponding to the entire node set. An entire node array can be declared by a NODES directive without "=*node-ref*".

executing node array A node array corresponding to the executing node set. An executing node array corresponding to the procedure can be declared by a NODES directive with the node reference "`*`".

parent node set The parent node set of a node set is the last executing node set, which encountered the innermost `task`, `loop`, or `array` construct that is being executed.

node number A unique number assigned to each node in a node set, which starts from one and corresponds to its position within the node set that is totally ordered.

2.8.3 Data Terminology

variable A named data storage block whose value can be defined and redefined during the execution of a program. Note that *variables* include array sections.

global data An array that is aligned with a template. Elements of a global data object are distributed onto nodes according to the distribution of the template. As a result, each node owns a part of the global data (called a *local section*), and can access it directly, but cannot access those on the other nodes.

local data Data that are not global. Each node owns a replica of a local data object, and can access it directly, but cannot access those on the other nodes. Note that the replicas of a local data object do not always have the same value.

replicated data Data whose storage is allocated on multiple nodes. A replicated data is either a local data object or a global data object replicated by an `align` directive.

distribution Assigning each element of a template to nodes in a node set in a specified manner. In a broad sense, it refers to assigning each element of an array, loop, etc.

alignment Associating each element of an array, loop, etc. with an element of the specified template. An element of the aligned array, loop, etc. is necessarily mapped to the same node as its associated element of the template.

local section A section of a global data object that is allocated as an array on each node at runtime. The local section of a global data object includes its shadow objects.

shadow An additional area of the local section of a distributed array, which is used to keep elements to be moved in from neighboring nodes.

2.8.4 Work Terminology

task A specific instance of executable codes that is defined by the `task` construct and executed by a node set specified by its `on` clause.

2.8.5 Communication and Synchronization Terminology

communication The movement of data between nodes. Communication in XcalableMP occurs only when the programmer instructs it explicitly using a global communication construct or a coarray reference.

reduction A procedure involving combining variables from each node in a specified manner and returning the result value. A reduction always involves communication. A reduction is specified by either the `on` clause of the `loop` construct or the `reduction` construct.

synchronization Synchronization is a mechanism to ensure that multiple nodes do not execute specific portions of a program at the same time. Synchronization between any number of nodes is specified by the `barrier` construct, and that between two nodes by the `post` and `wait` constructs.

asynchronous communication Communication that does not block, and which returns before it is complete. Thus, statements that follow it can overtake it. An asynchronous communication is specified by the `async` clause of global communication constructs or the `async` directive for a coarray reference.

2.8.6 Local-view Terminology

local alias An alias to the local section of a global data object, that is, a distributed array. A local alias can be used in XcalableMP programs in the same way as normal local data.

image An instance of an XcalableMP program corresponding to a respective node.

image set A totally ordered set of images.

image index An integer value that identifies an image in an image set, whose range is from one to the size of the image set.

entire image set The image set corresponding to the entire node set one to one in turn.

executing image set An image set corresponding to the executing node set one to one in turn.

The executing image set at the beginning of the program execution comprises the entire image set.

allocation image set An image set on which the coarray data object is allocated.

The allocation image set for a non-allocatable [F] or a static [C] coarray variable comprises the entire image set. Otherwise, the allocation image set for an allocatable [F] or an auto [C] coarray variable is the executing image set on which it is allocated unless it is specified by the `COARRAY` directive.

Chapter 3

Base Language Extensions in XcalableMP C

This chapter describes base language extensions in XcalableMP C that are not described in any other chapters.

3.1 Array Section Notation

Synopsis

The array section notation is a notation to describe a part of an array, which is adapted in Fortran.

Syntax

`[C]` *array-section* **is** *array-name*`[` { *triplet* | *int-expr* } `]`...

where *triplet* is:

[base] : *[length]* *[*: *step]*

Description

In XcalableMP C, the base language C is extended so that a part of an array, i.e., an array section, can be put in an *array assignment statement*, which is described in 3.2, and some XcalableMP constructs. An array section is built from a subset of the elements of an array, which is specified by this notation including at least one *triplet*.

When *step* is positive, the *triplet* specifies a set of subscripts that is a regularly spaced integer sequence of length *length* beginning with *base* and proceeding in increments of *step* up to the largest. When *step* is negative, the *triplet* specifies a set of subscripts that is a regularly spaced integer sequence of length *length* beginning with *base* and proceeding in increments of *step* down to the smallest.

When *base* is omitted, it is assumed to be 0. When *length* is omitted, it is assumed to account for the remainder of the array dimension. When *step* is omitted, it is assumed to be 1.

An array section can be considered as a virtual array containing the set of elements from the original array, which is determined by all possible subscript lists that are specified by the sequence of *triplets* or *int-expr*'s in square brackets.

Restrictions

- `[C]` Each of *base*, *length*, and *step* must be an integer expression.
- `[C]` *length* must be greater than zero.
- `[C]` *step* must not be zero.

Example

Assuming that an array `A` is declared by the following statement,

```
int A[100];
```

some array sections can be specified as follows:

`A[10:10]`	array section of 10 elements from `A[10]` to `A[19]`
`A[10:]`	array section of 90 elements from `A[10]` to `A[99]`
`A[:10]`	array section of 10 elements from `A[0]` to `A[9]`
`A[10:5:2]`	array section of 5 elements from `A[10]` to `A[18]` by step 2
`A[:]`	the whole of `A`

3.2 Array Assignment Statement

Synopsis

An array assignment statement copies a value into each element of an array section.

Syntax

`[C]` *array-section* `[:[`*int-expr*`]...] =` *expression*`;`

Description

The value of each element of the result of the right-hand side expression is assigned to the corresponding element of the array section on the left-hand side. When an operator or an elemental function (see section 7.8) is applied to array sections in the right-hand side expression, it is evaluated to an array section that has the same shape as that of the operands or arguments, and each element of which is the result of the operator or function applied to the corresponding element of the operands or arguments. A scalar object is assumed to be an array section that has the same shape as that of the array section(s), and where each element has its value.

Note that an array assignment is a statement, and therefore cannot appear as an expression in any other statements.

Restrictions

- `[C]` any array section appearing in the right-hand side expression and the left-hand side must have the same shape, i.e., the same number of dimensions and size of each dimension.
- `[C]` If *array-section* on the left-hand side is followed by "`:[`*int-expr*`]...", it must be a coarray.

Examples

An array assignment statement in the fourth line copies the elements `B[0]` through `B[4]` into the elements `A[5]` through `A[9]`.

XcalableMP C

```
int A[10];
int B[5];
    ...
A[5:5] = B[0:5];
```

3.3 Built-in Functions for Array Section

Some built-in functions are defined that can accept one or more array sections as arguments. In addition, some of them are array-valued. Such array-valued functions can appear in the right-hand side of an array assignment statement, and should be preceded by the `array` directive if the array section is distributed.

All of the built-in functions for array sections are described in Sections 7.8 and 7.9.

3.4 Pointer to Global Data

3.4.1 Name of Global Array

The name of a global array is considered to represent an abstract entity in the XcalableMP language. It is not interpreted as the pointer to the array, while the name of a local array is.

However, the name of a global array that appears in an expression is evaluated to the base address of its local section on each node. The pointer can be operated on each node as if it were a normal (local) pointer.

3.4.2 Address-of Operator

The result of the address-of operator ("`&`") applied to an element of a global array is the pointer to the corresponding element of its local section. Note that the value of the result pointer is defined only on the node that owns the element. The pointer can be operated on the node as if it were a normal (local) pointer.

As a result, for a global array `a`, `a` and `&a[0]` are not always evaluated to the same value.

3.5 Dynamic Allocation of Global Data

In XcalableMP C, it is possible to allocate global arrays at runtime. Such an allocation is done by performing the following steps.

1. Declare a pointer to an object of the type of the global array to be allocated.
2. Align the pointer with a template as if it were an array.
3. Allocate a block of memory of the global size using the `xmp_malloc` library procedure, and assign the return value to the pointer on each node.

XcalableMP C

```
#pragma nodes p(NP1,NP2)
#pragma xmp template t(:,:)
#pragma xmp distribute t(block,block) onto p

float (*pa)[N2];
#pragma xmp align pa[i][j] with t(i,j)
#pragma xmp template_fix t(0:N1-1,0:N2-1)

pa = (float (*)[N2])xmp_malloc(xmp_desc_of(pa), N1, N2);
```

3.6 Descriptor-of Operator

The descriptor-of operator ("`xmp_desc_of`") is introduced as a built-in operator in XcalableMP C.

The result of the descriptor-of operator applied to XcalableMP entities such as node arrays, templates, and global arrays is their *descriptor*, which can be used in various ways, including as an argument of some inquiry procedures. The type of the result, `xmp_desc_t`, is implementation-defined, and is defined in the `xmp.h` header file in XcalableMP C.

For the `xmp_desc_of` intrinsic function in XcalableMP Fortran, refer to section 7.1.1.

Chapter 4

Directives

This chapter describes the syntax and behavior of XcalableMP directives. In this document, the following notation is used to describe XcalableMP directives.

`xxx`	`type-face` characters are used to indicate literal-type characters.
xxx...	If the line is followed by "...", then xxx can be repeated.
[xxx]	*xxx* is optional.
▉	The syntax rule continues.
`[F]`	The following lines are effective only in XcalableMP Fortran.
`[C]`	The following lines are effective only in XcalableMP C.

4.1 Directive Format

4.1.1 General Rule

In XcalableMP Fortran, XcalableMP directives are specified using special comments that are identified by unique sentinels `!$xmp`. An XcalableMP directive follows the rules for comment lines of either the Fortran free or fixed source form, depending on the source form of the surrounding program unit[1]. XcalableMP Fortran directives are case insensitive.

`[F]` `!$xmp` *directive-name clause*

In XcalableMP C, XcalableMP directives are specified using the `#pragma` mechanism provided by the C standards. XcalableMP C directives are case-sensitive.

`[C]` `#pragma xmp` *directive-name clause*

Directives are classified as *declarative directives* and *executable directives*.

The declarative directive is a directive that may only be placed in a declarative context. A declarative directive has no associated executable user code. The scope rule of declarative directives obeys that of the declaration statements in the base language. For example, in XcalableMP Fortran, a node array declared by a `nodes` directive is visible only within either the program unit, the derived-type declaration, or the interface body that immediately surrounds the directives, unless it is overridden in the inner blocks or is use or host associated. Further, in XcalableMP C, a node array declared by a `nodes` directive is visible only in the range from the

[1]Consequently, the rules of comment lines that an XcalableMP directive follows are the same as the ones followed by an OpenMP directive.

declaring point to the end of the block when placed within a block, or of the file when placed outside any blocks, unless overridden in the inner blocks.

Note that in XcalableMP Fortran, node arrays and templates in other scoping units are accessible by use or host association.

The following directives are declarative directives.

- `nodes`
- `template`
- `distribute`
- `align`
- `shadow`
- `coarray`

The executable directives are placed in an executable context. A stand-alone directive is an executable directive that has no associated user code, such as a `barrier` directive. An executable directive and its associated user code make up an XcalableMP construct, as in the following format:

[F] `!$xmp` *directive-name clause ...*
structured-block

[C] `#pragma xmp` *directive-name clause ...*
structured-block

Note that in XcalableMP Fortran, a corresponding `end` directive is required for some executable directives such as `task` and `tasks`, and in XcalableMP C, the associated statement can be a compound one.

The following directives are executable directives.

- `template_fix`
- `task`
- `tasks`
- `loop`
- `array`
- `reflect`
- `gmove`
- `barrier`
- `reduction`
- `bcast`
- `wait_async`

4.1.2 Combined Directive

Synopsis

For XcalableMP Fortran, multiple attributes can be specified by one combined declarative directive, which is analogous to type declaration statements in Fortran using the "`::`" punctuation.

Syntax

`[F]` `!$xmp` *combined-directive* **is** *combined-attribute [, combined-attribute]...* `::` *combined-decl [, combined-decl]...*

combined-attribute is one of:

`nodes`
`template`
`distribute (`*dist-format [, dist-format]...* `)` `onto` *nodes-name*
`align (` *align-source [, align-source]...* `)` `with` *template-name* `(`*align-subscript [, align-subscript]...* `)`
`shadow (` *shadow-width [, shadow-width]...* `)`
`dimension (` *explicit-shape-spec [, explicit-shape-spec]...* `)`

and *combined-decl* is one of:

nodes-decl
template-decl
array-name

Description

A combined directive is interpreted as if an object corresponding to each *combined-decl* is declared in a directive corresponding to each *combined-attribute*, where all restrictions of each directive, in addition to the following ones, are applied.

Restrictions

- The same kind of *combined-attribute* must not appear more than once in a given *combined-directive*.
- If the `nodes` attribute appears in a *combined-directive*, each *combined-decl* must be a *nodes-decl*.
- If the `template` or `distribute` attribute appears in a *combined-directive*, each *combined-decl* must be a *template-decl*.
- If the `align` or `shadow` attribute appears in a *combined-directive*, each *combined-decl* must be an *array-name*.
- If the `dimension` attribute appears in a *combined-directive*, any object to which it applies must be declared using either the `template` or the `nodes` attribute.

4.2 `nodes` Directive

Synopsis

The `nodes` directive declares a named node array.

Syntax

[F] `!$xmp nodes` *nodes-decl [, nodes-decl]...*

[C] `#pragma xmp nodes` *nodes-decl [, nodes-decl]...*

where *nodes-decl* is one of:

nodes-name `(` *nodes-spec [, nodes-spec]...* `)`
nodes-name `(` *nodes-spec [, nodes-spec]...* `) =` *nodes-ref*
[C] *nodes-name* `[` *nodes-spec* `]` *[* `[` *nodes-spec* `]`*...]*
[C] *nodes-name* `[` *nodes-spec* `]` *[* `[` *nodes-spec* `]`*...]* `=` *nodes-ref*

and *nodes-spec* must be one of:

int-expr
`*`

Description

The `nodes` directive declares a node array that corresponds to a node set.

The first and third forms of the `nodes` directive are used to declare a node array that corresponds to the entire node set. The second and fourth forms are used to declare a node array, each element of which is assigned to the node of the node set specified by *nodes-ref* at the corresponding position of its elements order. In the first and second forms, which use parentheses, the element order of the declared node array is based on Fortran ' s. In the third and fourth forms, which use square brackets, the element order of the declared node array is based on C ' s.

If *node-size* in the last dimension is "`*`" in the first and second forms, or if that in the first dimension is "`*`" in the third and fourth forms, then the size of the node array is automatically adjusted according to the total size of either the entire node set in the first and third forms or the referenced node set in the second and fourth forms.

Restrictions

- *nodes-name* must not conflict with any other local name in the same scoping unit.
- *nodes-spec* can be "`*`" only in the last dimension in the first and second forms, and *nodes-spec* can be "`*`" only in the first dimension in the third and fourth forms.
- *nodes-ref* must not reference *nodes-name* either directly or indirectly.
- If no *nodes-spec* is "`*`", then the product of all *nodes-spec* must be equal to the total size of the entire node set in the first and third forms, or the referenced node set in the second and fourth forms.
- *nodes-subscript* in *nodes-ref* must not be "`*`".

Examples

The following are examples of the first and the third forms that appears in the main program. Because the node array `p`, which corresponds to the entire node set, is declared to be of size 16, this program must be executed by 16 nodes.

XcalableMP Fortran

```
      program main
!$xmp nodes p(16)
!$xmp nodes q(4,*)
!$xmp nodes r(8)=p(3:10)
!$xmp nodes z(2,3)=p(1:6)
      ...
      end program
```

XcalableMP C

```
int main() {
#pragma xmp nodes p[16]
#pragma xmp nodes q[*][4]
#pragma xmp nodes r[8]=p[2:8]
#pragma xmp nodes z[3][2]=p[0:6]
    ...
}
```

The following are examples of a node declaration in a procedure. Because `p` is declared in the second and fourth forms to have a size of 16 and corresponds to the executing node set, the invocation of the `foo` function must be executed by 16 nodes. The node array `q` is declared in the first and third forms, and corresponds to the entire node set. The node array `r` is declared as a subset of `p`, and `x` as a subset of `q`.

XcalableMP Fortran

```
      function foo()
!$xmp nodes p(16)=*
!$xmp nodes q(4,*)
!$xmp nodes r(8)=p(3:10)
!$xmp nodes x(2,3)=q(1:2,1:3)
      ...
      end function
```

XcalableMP C

```
void foo(){
#pragma xmp nodes p[16]=*
#pragma xmp nodes q[*][4]
#pragma xmp nodes r[8]=p[2:8]
#pragma xmp nodes x[3][2]=q[0:3][0:2]
    ...
}
```

4.2.1 Node Reference

Synopsis

The node reference is used to reference a node set.

Syntax

A node reference *nodes-ref* is specified by either the name of a node array or the "`*`" symbol.

nodes-ref **is** *nodes-name [*`(` *nodes-subscript [*, *nodes-subscript]*... `)` *]*

`[C]` *nodes-ref* **is** *nodes-name [*`[` *nodes-subscript* `]` *[* `[` *nodes-subscript* `]`... *]]*

or `*`

where *nodes-subscript* must be one of:

int-expr
triplet
`*`

Description

A node reference by *nodes-name* represents a node set corresponding to the node array specified by the name or its subarray. It is totally ordered in Fortran's array element order in the first form, and in C's array element order in the second form. A node reference by "`*`" represents the executing node set.

Specifically, the "`*`" symbol that appears as *nodes-subscript* in a dimension of *nodes-ref* is interpreted by each node at runtime as its position (coordinate) in the dimension of the referenced

node array. Thus, a node reference `p(`s_1`, ..., `s_{k-1}`, *, `s_{k+1}`, ..., `s_n`)` is interpreted as `p(`s_1`, ..., `s_{k-1}`, `j_k`, `s_{k+1}`, ..., `s_n`)` on the node `p(`j_1`, ..., `j_{k-1}`, `j_k`, `j_{k+1}`, ..., `j_n`)`.

Note that "`*`" can be used only as the node reference in the `on` clause of some executable directives.

Examples

Assume that `p` is the name of a node array and that `m` is an integer variable.

- As a target node array in the `distribute` directive,

XcalableMP Fortran
```
!$xmp distribute a(block) onto p
```
XcalableMP C
```
#pragma xmp distribute a(block) onto p
```

- To specify the node array to which the declared node array corresponds in the second and fourth forms of the `nodes` directive,

XcalableMP Fortran
```
!$xmp nodes r(2,2,4) = p(1:4,1:4)
!$xmp nodes r(2,2,4) = p(1:16)
```
XcalableMP C
```
#pragma xmp nodes r[4][2][2] = p[0:4][0:4]
#pragma xmp nodes r[4][2][2] = p[0:16]
```

- To specify the node array that corresponds to the executing node set of a task in the `task` directive,

XcalableMP Fortran
```
!$xmp task on p(1:4,1:4)
!$xmp task on p(1:16)
!$xmp task on p(:,*)
!$xmp task on p(m)
```
XcalableMP C
```
#pragma xmp task on p[0:4][0:4]
#pragma xmp task on p[0:16]
#pragma xmp task on p[*][:]
#pragma xmp task on p[m]
```

- To specify the node array that corresponds to the executing node set in the `barrier` and the `reduction` directive,

XcalableMP Fortran
```
!$xmp barrier on p(5:8)
!$xmp reduction (+:a) on p(*,:)
```
XcalableMP C
```
#pragma xmp barrier on p[4:4]
#pragma xmp reduction (+:a) on p[:][*]
```

- To specify the source node and the node array that corresponds to the executing node set in the `bcast` directive,

XcalableMP Fortran
```
!$xmp bcast (b) from p(k) on p(:)
```
XcalableMP C
```
#pragma xmp (b) from p[k-1] on p[:]
```

4.3 Template and Data Mapping Directives

4.3.1 `template` Directive

Synopsis

The `template` directive declares a template.

Syntax

[F] `!$xmp template` *template-decl [, template-decl]...*

[C] `#pragma xmp template` *template-decl [, template-decl]...*

where *template-decl* is:

template-name `(` *template-spec [*, *template-spec]...* `)`
`[C]` *template-name* `[` *template-spec-c* `]` *[* `[` *template-spec-c* `]...` *]*

and *template-spec* must be one of:

[int-expr `:` *]* *int-expr*
`:`

and *template-spec-c* must be one of:

int-expr
`:`

Description

The `template` directive declares a template with the shape specified by the sequence of *template-spec*'s or *template-spec-c*'s. If every *template-spec* or *template-spec-c* is ":", then the shape of the template is initially undefined. This template must not be referenced until the shape is defined by a `template_fix` directive (see section 4.3.6) at runtime. If only *int-expr* is specified as *template-spec*, the default lower bound is one.

Restrictions

- *template-name* must not conflict with any other local name in the same scoping unit.
- Every *template-spec* must be either *[int-expr* `:` *]* *int-expr* or ":".
- Every *template-spec-c* must be either *int-expr* or ":".

4.3.2 Template Reference

Synopsis

The template reference expression specified in the `on` or the `from` clause of some directives is used to indirectly specify a node set.

Syntax

template-ref **is** *template-name [*`(` *template-subscript [*, *template-subscript]...* `)` *]*
`[C]` *template-ref* **is** *template-name [*`[` *template-subscript* `]` *[* `[` *template-subscript* `]...` *]]*

where *template-subscript* must be one of:

int-expr
triplet
`*`

Description

Being specified in the `on` or the `from` clause of some directives, the template reference refers to a subset of a node set in which the specified subset of the template resides.

Specifically, the "`*`" symbol that appeares as *template-subscript* in a dimension of *template-ref* is interpreted by each node at runtime as the indices of the elements in the dimension that reside in the node. "`*`" in a template reference is similar to "`*`" in a node reference.

Examples

Assume that `t` is a template.

- In the `task` directive, the executing node set of the task can be indirectly specified using a template reference in the `on` clause.

XcalableMP Fortran
```
!$xmp task on t(1:m,1:n)
!$xmp task on t
```

XcalableMP C
```
#pragma xmp task on t[0:n][0:m]
#pragma xmp task on t
```

- In the `loop` directive, the executing node set of each iteration of the following loop is indirectly specified using a template reference in the `on` clause.

XcalableMP Fortran
```
!$xmp loop (i) on t(i-1)
```

XcalableMP C
```
#pragma xmp loop (i) on t[i-1]
```

- In the `array` directive, the executing node set on which the associated array-assignment statement is performed in parallel is indirectly specified using a template reference in the `on` clause.

XcalableMP Fortran
```
!$xmp array on t(1:n)
```

XcalableMP C
```
#pragma xmp array on t[0:n]
```

- In the `barrier`, `reduction`, and `bcast` directives, the node set that is to perform the operation collectively can be indirectly specified using a template reference in the `on` clause.

XcalableMP Fortran
```
!$xmp barrier on t(1:n)
!$xmp reduction (+:a) on t(*,:)
!$xmp bcast (b) on t(1:n)
```

XcalableMP C
```
#pragma xmp barrier on t[0:n]
#pragma xmp reduction (+:a) on t[:][*]
#pragma xmp bcast (b) on t[0:n]
```

4.3.3 `distribute` Directive

Synopsis

The `distribute` directive specifies the distribution of a template.

Syntax

[F] `!$xmp distribute` *template-name* `(`*dist-format [, dist-format]...* `) onto` *nodes-name*

[C] `#pragma xmp distribute` *template-name* `(`*dist-format [, dist-format]...* `) onto` *nodes-name*

[C] `#pragma xmp distribute` *template-name* `[` *dist-format* `] [ [` *dist-format* `] ... ] onto` *nodes-name*

where *dist-format* must be one of:

`*`
`block` *[* `(` *int-expr* `)` *]*
`cyclic` *[* `(` *int-expr* `)` *]*
`gblock (` { `*` | *int-array* } `)`

Description

According to the specified distribution format, a template is distributed onto a specified node array. The dimension of the node array that appears in the `onto` clause corresponds, in order of left-to-right, to the dimension of the distributed template for which the corresponding *dist-format* is not "`*`".

Let `d` be the size of the dimension of the template, `p` be the size of the corresponding dimension of the node array, `ceiling` and `mod` be Fortran's intrinsic functions, and each of the arithmetic operators be that of Fortran. The interpretation of *dist-format* is as follows:

"`*`" The dimension is not distributed.

`block` Equivalent to `block(ceiling(d/p))`.

`block(n)` The dimension of the template is divided into contiguous blocks of size `n`, which are distributed onto the corresponding dimension of the node array. The dimension of the template is divided into `d/n` blocks of size `n`, and one block of size `mod(d,n)` if any, and each block is assigned sequentially to an index along the corresponding dimension of the node array. Note that if `k = p-d/n-1 > 0`, then there is no block assigned to the last `k` indices.

`cyclic` Equivalent to `cyclic(1)`.

`cyclic(n)` The dimension of the template is divided into contiguous blocks of size `n`, and these blocks are distributed onto the corresponding dimension of the node array in a round-robin manner.

`gblock(m)` `m` is referred to as a mapping array. The dimension of the template is divided into contiguous blocks so that the i'th block is of size `m(i)`, and these blocks are distributed onto the corresponding dimension of the node array.

If at least one `gblock(*)` is specified in *dist-format*, then the template is initially undefined and must not be referenced until the shape of the template is defined by `template_fix` directives at runtime.

Restrictions

- `[C]` *template-name* must be declared by a `template` directive that lexically precedes the directive.
- The number of *dist-format* that is not "`*`" must be equal to the rank of the node array specified by *nodes-name*.
- The size of the dimension of the template specified by *template-name* that is distributed by `block(n)` must be equal to or less than the product of the block size `n` and the size of the corresponding dimension of the node array specified by *nodes-name*.
- The array *int-array* in parentheses following `gblock` must be an integer one-dimensional array, and its size must be equal to the size of the corresponding dimension of the node array specified by *nodes-name*.
- Every element of the array *int-array* in parentheses following `gblock` must have a value of a nonnegative integer.

- The sum of the elements of the array *int-array* in parentheses following `gblock` must be equal to the size of the corresponding dimension of the template specified by *template-name*.

- `[C]` A `distribute` directive for a template must precede any of its references in the executable code in the block.

Examples

Example 1

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(64)
!$xmp distribute t(block) onto p
```

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[64]
#pragma xmp distribute t[block] onto p
```

The template `t` is distributed in `block` format, as shown in the following table.

p(1)	t(1:16)
p(2)	t(17:32)
p(3)	t(33:48)
p(4)	t(49:64)

p[0]	t[0:16]
p[1]	t[16:16]
p[2]	t[32:16]
p[3]	t[48:16]

Example 2

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(64)
!$xmp distribute t(cyclic(8)) onto p
```

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[64]
#pragma xmp distribute t[cyclic(8)] onto p
```

The template `t` is distributed in `cyclic` format of size eight, as shown in the following table.

p(1)	t(1:8) t(33:40)
p(2)	t(9,16) t(41:48)
p(3)	t(17,24) t(49:56)
p(4)	t(25,32) t(57:64)

p[0]	t[0:8] t[32:8]
p[1]	t[8:8] t[40:8]
p[2]	t[16:8] t[48:8]
p[3]	t[24:8] t[56:8]

Example 3

XcalableMP Fortran

```
!$xmp nodes p(8,5)
!$xmp template t(64,64,64)
!$xmp distribute t(*,cyclic,block) onto p
```

XcalableMP C

```
#pragma xmp nodes p[5][8]
#pragma xmp template t[64][64][64]
#pragma xmp distribute t[block][cyclic][*] onto p
```

The first dimension of the template `t` is not distributed. The second dimension is distributed onto the first dimension of the node array `p` in `cyclic` format. The third dimension is distributed onto the second dimension of `p` in `block` format. The results are as follows:

p(1,1)	t(1:64, 1:57:8, 1:13)
p(2,1)	t(1:64, 2:58:8, 1:13)
...	...
p(8,5)	t(1:64, 8:64:8, 53:64)

p[0][0]	t[0:13][0:8:8][0:64]
p[0][1]	t[0:13][1:8:8][0:64]
...	...
p[4][7]	t[52:12][7:8:8][0:64]

Note that the "64" in `template t` is not divisible by "5" in `node p`. Thus, the sizes of the blocks are different among nodes.

4.3.4 `align` Directive

Synopsis

The `align` directive specifies that an array is to be mapped in the same way as a specified template.

Syntax

[F] `!$xmp align` *array-name* `(` *align-source [, align-source]...* `)` `&`
`&` `with` *template-name* `(`*align-subscript [, align-subscript]...* `)`

[C] `#pragma xmp align` *array-name* `[`*align-source*`]` *[*`[`*align-source*`]`*]...* `&`
`&` `with` *template-name* `(`*align-subscript [, align-subscript]...* `)`
or
`&` `with` *template-name* `[`*align-subscript* `]` *[* `[` *align-subscript* `]`*...]*

where *align-source* must be one of:

scalar-int-variable
`*`
`:`

and *align-subscript* must be one of:

scalar-int-variable [{ `+` | `-` } *int-expr]*
`*`
`:`

Note that the variable *scalar-int-variable* that appears in *align-source* is referred to as an "align dummy variable" and *int-expr* appearing in *align-subscript* as an "align offset."

Description

The array specified by *array-name* is aligned with the template that is specified by *template-name* so that each element of the array indexed by the sequence of *align-sources* is aligned with the element of the template indexed by the sequence of *align-subscripts*, where *align-sources* and *align-subscripts* are interpreted as follows:

1. The first form of *align-source* and *align-subscript* represents an align dummy variable and an expression of it, respectively. The align dummy variable is considered to range over all valid index values in the corresponding dimension of the array.

2. The second form "`*`" of *align-source* and *align-subscript* represents a dummy variable (not an align dummy variable) that does not appear anywhere in the directive.

   - The second form of *align-source* is said to "collapse" the corresponding dimension of the array. As a result, the index along the corresponding dimension does not affect the determination of the alignment.
   - The second form of *align-subscript* is said to "replicate" the array. Each element of the array is replicated, and is aligned to all index values in the corresponding dimension of the template.

3. The third form of *align-source* and the matching *align-subscript* represents the same align dummy variable whose range spans all valid index values in the corresponding dimension

of the array. The matching of colons (":") in the sequence of *align-sources* and *align-subscripts* is determined as follows:

- `[F]` Colons in the sequence of *align-sources* and those in the sequence of *align-subscripts* are matched in corresponding left-to-right order, where any *align-source* and *align-subscript* that is not a colon is ignored.
- `[C]` Colons in the sequence of *align-sources* in right-to-left order, and those in the sequence of `(`*align-subscript*`)`'s in left-to-right order are matched, or those in the sequence of `[`*align-subscript*`]`'s in right-to-left order are matched, where any *align-source* and *align-subscript* that is not a colon is ignored.

In XcalableMP C, an `align` directive for a dummy argument can be placed either outside the function body (as in the old style of C) or in it (as in the ANSI style).

Restrictions

- `[C]` *array-name* must be declared by a declaration statement that lexically precedes the directive.
- An align dummy variable may appear at most once in the sequence of *align-sources*.
- An align dummy variable may appear at most once in the sequence of *align-subscripts*.
- An *align-subscript* may contain at most one occurrence of an align dummy variable.
- The *int-expr* in an *align-subscript* may not contain any occurrence of an align dummy variable.
- The sequence of *align-sources* must contain exactly as many colons as contained by the sequence of *align-subscripts*.
- `[F]` The array specified by *array-name* must not appear as an *equivalence-object* in an `equivalence` statement.
- `[C]` An `align` directive for an array must precede any of its appearances in the executable code in the block.
- `[F]` The array specified by *array-name* shall not be initially defined.
- `[C]` The array specified by *array-name* shall not be initialized through an *initializer*.

Examples

Example 1

XcalableMP Fortran

```
!$xmp align a(i) with t(i)
```

XcalableMP C

```
#pragma xmp align a[i] with t[i]
```

In XcalableMP Fortran, the array element `a(i)` is aligned with the template element `t(i)`. In XcalableMP C, the array element `a[i]` is aligned with the template element `t[i]`. These are equivalent to the following codes.

XcalableMP Fortran

```
!$xmp align a(:) with t(:)
```

XcalableMP C

```
#pragma xmp align a[:] with t[:]
```

Example 2

XcalableMP Fortran	XcalableMP C
`!$xmp align a(*,j) with t(j)`	`#pragma xmp align a[j][*] with t[j]`

In XcalableMP Fortran, the subarray `a(:,j)` is aligned with the template element `t(j)`. Note that the first dimension of `a` is collapsed. In XcalableMP C, the subarray `a[j][:]` is aligned with the template element `t[j]`. Note that the second dimension of `a` is collapsed.

Example 3

XcalableMP Fortran	XcalableMP C
`!$xmp align a(j) with t(*,j)`	`#pragma xmp align a[j] with t[j][*]`

In XcalableMP Fortran, the array element `a(j)` is replicated and aligned with each template element of `t(:,j)`. In XcalableMP C, the array element `a[j]` is replicated and aligned with each template element of `t[j][:]`.

Example 4

XcalableMP Fortran	XcalableMP C

`!$xmp template t(n1,n2)`
`real a(m1,m2)`
`!$xmp align a(*,j) with t(*,j)`
	`#pragma xmp template t[n2][n1]`
	`double a[m2][m1]`
	`#pragma xmp align a[j][*] with t[j][*]`

In XcalableMP Fortran, the subarray `a(:,j)` is aligned with each template element of `t(:,j)`. In XcalableMP C, the subarray `a[j][:]` is aligned with each template element of `t[j][:]`.

By replacing "`*`" of the array `a` and "`*`" of the template `t` with a dummy variable `i` and `k`, respectively, this alignment can be interpreted as the following mapping.

`[F]` $a(i,j) \rightarrow t(k,j) \mid (i,j,k) \in (1:n1,\ 1:n2,\ 1:m1)$
`[C]` $a[j][i] \rightarrow t[j][k] \mid (i,j,k) \in (0:n1,\ 0:n2,\ 0:m1)$

4.3.5 `shadow` Directive

Synopsis

The `shadow` directive allocates the shadow area for a distributed array.

Syntax

`[F]` `!$xmp shadow` *array-name* `(` *shadow-width [, shadow-width]...* `)`

`[C]` `#pragma xmp shadow` *array-name* `[`*shadow-width*`]`*[*`[`*shadow-width*`]`*]...*

where *shadow-width* must be one of:

int-expr
int-expr : *int-expr*
`*`

Description

The `shadow` directive specifies the width of the shadow area of an array specified by *array-name*, which is used to communicate the neighbor element of the block of the array. When *shadow-width* is of the form "*int-expr* : *int-expr*," the shadow area of the width specified by the first *int-expr* is added at the lower bound, and that specified by the second one is added at the upper

bound in the dimension. When *shadow-width* is of the form *int-expr*, the shadow area of the same width specified is added at both the upper and lower bounds in the dimension. When *shadow-width* is of the form "`*`", the entire area of the array is allocated on each node, and the area that it does not own is regarded as a shadow. This type of shadow is sometimes referred to as a "full shadow."

Note that the shadow area of a multi-dimensional array includes "obliquely-neighboring" elements, which are owned by the node whose indices are different in more than one dimension, and that the shadow area can also be allocated at the global lower and upper bounds of an array.

The data stored in the storage area declared by the `shadow` directive is referred to as a *shadow object*. A shadow object represents an element of a distributed array, and corresponds to the data object that represents the same element as itself. The corresponding data object is referred to as the *reflection source* of the shadow object.

Restrictions

- `[C]` *array-name* must be declared by a declaration statement that lexically precedes the directive.
- The value specified by *shadow-width* must be a nonnegative integer.
- The number of *shadow-width* must be equal to the number of dimensions (or rank) of the array specified by *array-name*.
- `[C]` A `shadow` directive for an array must precede any of its appearances in the executable code in the block.

Example

XcalableMP Fortran

```
!$xmp nodes p(4,4)
!$xmp template t(64,64)
!$xmp distribute t(block,block) onto p

      real a(64,64)
!$xmp align a(i,j) with t(i,j)
!$xmp shadow a(1,1)
```

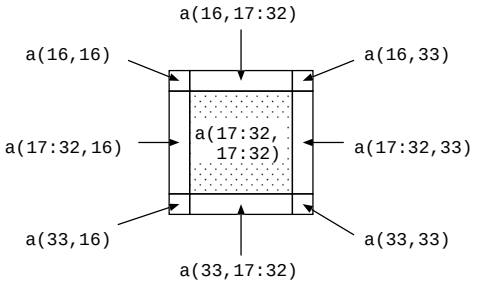

Figure 4.1: Example showing shadow of a two-dimensional array.

The node `p(2,2)` has `a(17:32,17:32)` as a data object, and `a(16,16)`, `a(17:32,16)`, `a(33,16)`, `a(16,17:32)`, `a(33,17:32)`, `a(16,33)`, `a(17:32,33)`, and `a(33,33)` as shadow objects (Figure 4.1). Among them, `a(16,16)`, `a(33,16)`, `a(16,33)`, and `a(33,33)` are "obliquely-neighboring" elements of `p(2,2)`.

4.3.6 `template_fix` Construct

Synopsis

This construct fixes the shape and/or the distribution of an undefined template.

Syntax

[F] `!$xmp template_fix` ▉
▉ *[*`(` *dist-format [, dist-format]...* `)`*] template-name [*`(`*template-spec [, template-spec]...* `)`*]*

[C] `#pragma xmp template_fix` ▉
▉ *[*`(` *dist-format [, dist-format]...*`)`*] template-name [*`(`*template-spec [, template-spec]...* `)`*]*
or
▉ *[*`[` *dist-format [, dist-format]...*`]`*] template-name* `[` *template-spec* `]` *[* `[`*template-spec* `]`*...]*

where *template-spec* is:

[int-expr :] int-expr

and *dist-format* is one of:

`*`
`block` *[*`(` *int-expr* `)`*]*
`cyclic` *[*`(` *int-expr* `)`*]*
`gblock (` *int-array* `)`

Description

The `template_fix` construct fixes the shape and/or the distribution of the template that is initially undefined, by specifying the sizes and/or the distribution format of each dimension at runtime. Arrays that are aligned with an initially undefined template must be allocatable arrays, in XcalableMP Fortran, or a pointer (see Section 3.5), in XcalableMP C, which cannot be allocated until the template is fixed by the `template_fix` construct. No constructs that have such a template in their `on` clause should be encountered until the template is fixed by the `template_fix` construct. Any undefined template can be fixed only once by the `template_fix` construct in its scoping unit.

The meaning of the sequence of *dist-formats* is the same as that in the `distribute` directive.

Restrictions

- When a node encounters a `template_fix` construct at runtime, the template specified by *template-name* must be undefined.

- If the sequence of *dist-formats* exists in a `template_fix` construct, it must be identical to the sequence of *dist-formats* in the `distribute` directive for the template specified by *template-name*, except for *int-array* specified in the parenthesis immediately after `gblock`.

- Either the sequence of *dist-formats* or the sequence of *template-spec*'s should be given.

Example

XcalableMP Fortran

```
!$xmp template t(:)
!$xmp distribute t(gblock(*))
real, allocatable :: a(:)
!$xmp align a(i) with t(i)
...
N = ...
M(...) = ...
...
!$xmp template_fix(gblock(M)) t(N)
...
allocate (a(N))
```

XcalableMP C

```
#pragma xmp template t[:]
#pragma xmp distribute t[gblock(*)]
double *a;
#pragma xmp align a[i] with t[i]
...
N = ...;
M[] = {...};
...
#pragma xmp template_fix[gblock(M)] t[N]
...
a = xmp_malloc(xmp_desc_of(a), N);
```

Because the shape is `t(:)` or `t[:]` and the distribution format is `gblock(*)`, the template `t` is initially undefined. The allocatable array `a` is aligned with `t`. After the size `N` and the mapping array `M` is defined, `t` is fixed by the `template_fix` construct and `a` is allocated.

In XcalableMP C, it is possible to allocate global arrays at runtime only when they are one-dimensional. Such an allocation is done by perfoming the following steps.

1. Declare a pointer to an object of the type of the global array to be allocated.
2. Align the pointer with a template as if it were a one-dimensional array.
3. Allocate a storage of the global size with the function `xmp_malloc()` and assign the result value to the pointer on each node.

The functions `xmp_desc_of()` and `xmp_malloc()` are described in section 3.6 and 7.5.1, respectively.

4.4 Work Mapping Construct

4.4.1 `task` Construct

Synopsis

The `task` construct defines a task that is executed by a specified node set.

Syntax

[F] `!$xmp task on` {*nodes-ref* | *template-ref*}
structured-block
`!$xmp end task`

[C] `#pragma xmp task on` {*nodes-ref* | *template-ref*}
structured-block

Description

When a node encounters a `task` construct at runtime, it executes the associated block (called a *task*) if it is included by the node set specified by the `on` clause; otherwise, it skips the execution of the block.

Unless a `task` construct is surrounded by a `tasks` construct, *nodes-ref* or *template-ref* in the `on` clause is evaluated by the executing node set at the start of the task; otherwise, *nodes-ref* and *template-ref* of the `task` construct are evaluated by the executing node set at the entry of the `tasks` construct that immediately surrounds it. The current executing node set is set to be that specified by the `on` clause at the entry of the `task` construct, and it is rewound to the last one at the exit.

Restrictions

- The node set specified by *nodes-ref* or *template-ref* in the `on` clause must be a subset of the parent node set.

Example

Example 1 In XcalableMP Fortran, copies of variables `a` and `b` are replicated on nodes `nd(1)` through `nd(8)`. A task defined by the `task` construct is executed only on `nd(1)`, and defines the copies of `a` and `b` on a node `nd(1)`. The copies on nodes `nd(2)` through `nd(8)` are not defined.

In XcalableMP C, copies of variables `a` and `b` are replicated on nodes `nd[0]` through `nd[7]`. A task defined by the `task` construct is executed only on `nd[0]`, and defines the copies of `a` and `b` on a node `nd[0]`. The copies on nodes `nd[1]` through `nd[7]` are not defined.

XcalableMP Fortran

```
!$xmp nodes nd(8)
!$xmp template t(100)
!$xmp distribute t(block) onto nd

      real a, b;

!$xmp task on nd(1)
      read(*,*) a
      b = a*1.e-6
!$xmp end task
```

XcalableMP C

```
#pragma xmp nodes nd[8]
#pragma xmp template t[100]
#pragma xmp distribute t[block] onto nd

    float a, b;

#pragma xmp task on nd[0]
    {
        scanf ("%f", &a);
        b = a*1.e-6;
    }
```

Example 2 According to the `on` clause with a template reference, an assignment statement in the `task` construct is executed by the owner of the array element `a(:,j)` or `a[j][:]`.

XcalableMP Fortran

```
!$xmp nodes nd(8)
!$xmp template t(100)
!$xmp distribute t(block) onto nd

      integer i,j
      real a(200,100)
!$xmp align a(*,j) with t(j)

      i = ...
      j = ...

!$xmp task on t(j)
      a(i,j) = 1.0
!$xmp end task
```

XcalableMP C

```
#pragma xmp nodes nd[8]
#pragma xmp template t[100]
#pragma xmp distribute t(block) onto nd

    int i,j;
    float a[100][200];
#pragma align a[j][*] with t[j]

    i = ...;
    j = ...;

#pragma xmp task on t[j]
    a[j][i] = 1.0;
}
```

4.4.2 `tasks` Construct

Synopsis

The `tasks` construct is used to instruct the executing nodes to execute the multiple tasks that it surrounds in an arbitrary order.

Syntax

[F]
```
!$xmp tasks
task-construct
...
!$xmp end tasks
```

[C]
```
#pragma xmp tasks
{
    task-construct
    ...
}
```

Description

`task` constructs surrounded by a `tasks` construct are executed in arbitrary order without implicit synchronization at the start of each task. As a result, if there are no overlaps between the executing node sets of the adjacent tasks, they can be executed in parallel.

nodes-ref or *template-ref* of each task immediately surrounded by a `tasks` construct is evaluated by the executing node set at the entry of the `tasks` construct.

No implicit synchronization is performed at the start and end of the `tasks` construct.

Example

Example 1 Three instances of subroutine `task1` are concurrently executed by node sets `p(1:500)`, `p(501:800)`, and `p(801:1000)`.

XcalableMP Fortran

```
      subroutine caller
!$xmp nodes p(1000)
!$xmp template tp(100)
!$xmp distribute t(block) onto p
      real a(100,100)
!$xmp align a(*,k) with t(k)
      ...
!$xmp tasks
!$xmp  task on p(1:500)
        call task1(a)
!$xmp  end task
!$xmp  task on p(501:800)
        call task1(a)
!$xmp  end task
!$xmp  task on p(801:1000)
        call task1(a)
!$xmp  end task
!$xmp end tasks
      ...
      end subroutine
```

XcalableMP Fortran

```
      subroutine task1(a)
      ...
!$xmp nodes q(*)=*

!$xmp nodes p(1000)
!$xmp distribute t(block) onto p
      real a(100,100)
!$xmp align a(*,k) with t(k)
      ...
      end subroutine
```

Example 2 The first node `p(1)` executes the first and second tasks, the final `node p(8)` the second and the third tasks, and the other nodes `p(2)` through `p(7)` only the second task.

XcalableMP Fortran

```
!$xmp nodes p(8)
!$xmp template t(100)
!$xmp distribute t(block) onto p
      real a(100)
!$xmp align a(i) with t(i)
      ...
!$xmp tasks

!$xmp task on t(1)
      a(1) = 0.0
!$xmp end task

!$xmp task on t(2:99)
!$xmp loop on t(i)
      do i=2,99
        a(i) = foo(i)
      enddo
!$xmp end task

!$xmp task on t(100)
      a(100) = 0.0
!$xmp end task
```

```
!$xmp end tasks
```

4.4.3 `loop` Construct

Synopsis

The `loop` construct specifies that each iteration of the following loop is executed by a node set that is specified by the `on` clause, so the iterations are distributed among nodes and executed in parallel.

Syntax

[F] `!$xmp loop` *[* `(` *loop-index [, loop-index]...* `)` *]* `on` {*nodes-ref* | *template-ref*} ▮
▮ *[* `expand(` *expand-width [, expand-width]...* `)` *]* ▮
▮ *[* `margin(` *margin-width [, margin-width]...* `)` *]* ▮
▮ *[reduction-clause]...*
do-loops

[C] `#pragma xmp loop` *[* `(` *loop-index [, loop-index]...* `)` *]* `on` {*nodes-ref* | *template-ref*} ▮
▮ *[* `expand(` *expand-width [, expand-width]...* `)` *]* ▮
▮ *[* `margin(` *margin-width [, margin-width]...* `)` *]* ▮
▮ *[reduction-clause]...*
for-loops

where *expand-width* and *margin-width* must be one of:

[`/unbound/`*] int-expr*
[`/unbound/`*] int-expr* : *int-expr*

reduction-clause is:

`reduction(` *reduction-kind* : *reduction-spec [, reduction-spec]...* `)`

reduction-kind is one of:

```
[F]  +
     *
     -
     .and.
     .or.
     .eqv.
     .neqv.
     max
     min
     iand
     ior
     ieor
     firstmax
     firstmin
     lastmax
     lastmin

[C]  +
     *
     -
     &
     |
     ^
     &&
     ||
     max
     min
     firstmax
     firstmin
     lastmax
     lastmin
```

and *reduction-spec* is:

reduction-variable [`/` *location-variable [*, *location-variable]...* `/` *]*

Description

A `loop` directive is associated with a loop nest consisting of one or more tightly nested loops that follow the directive, and it distributes the execution of their iterations onto the node set specified by the `on` clause.

The sequence of *loop-indexes* in parenthesis denotes an index of an iteration of the loop nests. If a control variable of a loop does not appear in the sequence, it is assumed that each of its possible values is specified in the sequence. The sequence can be considered to denote a set of indices of iterations. When the sequence is omitted, it is assumed that the control variables of all the loops in the associated loop nests are specified.

When a *template-ref* is specified in the `on` clause, the associated loop is distributed so that the iteration (set) indexed by the sequence of *loop-indexes* is executed by the node onto which a template element specified by the *template-ref* is distributed.

When a *nodes-ref* is specified in the `on` clause, the associated loop is distributed so that the iteration (set) indexed by the sequence of *loop-indexes* is executed by a node specified by the

nodes-ref.

In addition, the executing node set is updated to the node set specified by the `on` clause at the beginning of every iteration, and it is restored to the last one at the end of it.

When a *reduction-clause* is specified, a reduction operation of the kind specified by *reduction-kind* for a variable specified by *reduction-variable* is executed just after the execution of the loop nest.

When the `expand` clause is specified, and is of the form "*int-expr* : *int-expr*" in a dimension, the first *int-expr* is subtracted from the local lower bound in that dimension, and the second one is added to the local upper bound. When the `expand` clause is specified, and is of the form *int-expr*, the *int-expr* is subtracted from the local lower bound in that dimension, and is added to the local upper bounds. However, an "expanded" local iteration space does not spread out of the original global iteration space unless the `/unbound/` modifier is specified in *expand-width*.

When the `margin` clause is specified, the loop is transformed so that its local iteration space, *margin*, is:

$$margin = expand \triangle orig$$

where *expand* is a local iteration space when an `expand` clause with the same argument(s) is specified, *orig* is a local iteration space when neither *expand* nor *margin*, and $\triangle$ is the symmetric difference operator.

(Advice to programmers and implementers) Using the `expand` and `margin` clauses and asynchronous communication, programmers can overlap computation and communication as in the code left below. It is recommended for the implementation to support an extension that is a syntactic sugar for those sequence of constructs, such as the `peel_and_wait` clause in the code immediately following.

XcalableMP Fortran

```
!$xmp reflect (a) async(10)

!$xmp loop (i,j) on t(i,j)
!$xmp+               expand(-1,-1)
      do j = 1, 16
          do i = 1, 16
              ...
          end do
      end do

!$xmp wait_async (10)

!$xmp loop (i,j) on t(i,j)
!$xmp+               margin(-1,-1)
      do j = 1, 16
          do i = 1, 16
              ...
          end do
      end do
```

XcalableMP Fortran

```
!$xmp reflect (a) async(10)

!$xmp loop (i,j) on t(i,j)
!$xmp+   peel_and_wait(10, -1,-1)
      do j = 1, 16
          do i = 1, 16
              ...
          end do
      end do
```

The reduction operation that is executed, except in cases with *reduction-kind* of `FIRSTMAX`, `FIRSTMIN`, `LASTMAX`, or `LASTMIN`, is equivalent to the `reduction` construct with *reduction-kind* of "`+`" for "`-`" in the clause and the same *reduction-kind* for the other kinds, the same *reduction-variable*, and an `on` clause obtained from that of the `loop` directive by replacing each *loop-index*

in the *nodes-ref* or the *template-ref* with a triplet representing the range of its value. As an example, the two codes below are therefore equivalent.

XcalableMP Fortran

```
!$xmp loop (j) on t(:,j)
!$xmp+              reduction(op:s)
      do j = js, je
        ...
        do i = 1, N
          s = s op a(i,j)
        end do
        ...
      end do
```

XcalableMP Fortran

```
! Initialize s_tmp to the identity
! element of the op operator
      s_tmp = ...

!$xmp loop (j) on t(:,j)
      do j = js, je
        ...
        do i = 1, N
          s_tmp = s_tmp op a(i,j)
        end do
        ...
      end do

!$xmp reduction(op:s_tmp)
!$xmp+                on t(*,js:je)

      s = s op s_tmp
```

In particular, for the reduction kinds of `FIRSTMAX`, `FIRSTMIN`, `LASTMAX`, and `LASTMIN`, in addition to a corresponding `MAX` or `MIN` reduction operation, the *location-variables* are set after executing the `loop` construct as follows:

- For `FIRSTMAX` and `FIRSTMIN`, they are set to their values at the end of the *first* iteration in which the *reduction-variable* takes the value of the reduction result, where *first* refers to the first position in the sequential order in which iterations of the associated loop nest were executed without parallelization.
- For `LASTMAX` and `LASTMIN`, they are set to their values at the end of the *last* iteration in which the *reduction-variable* takes the value of the reduction result, where *last* refers to the last position in the sequential order in which iterations of the associated loop nest were executed without parallelization.

Restrictions

- *loop-index* must be a control variable of a loop in the associated loop nest.
- A control variable of a loop can appear as *loop-index* at most once.
- The node set specified by *nodes-ref* or *template-ref* in the `on` clause must be a subset of the parent node set.
- The template specified by *template-ref* must be fixed before the `loop` construct is executed.
- The `loop` construct is global, which means that it must be executed by all of the executing nodes with the same values for each local variable referenced in the directive, and the lower bound, upper bound, and step of the associated loop.
- Either of the `expand` or `margin` clause, if any, can be specified.
- The number of *expand-width*, if any, must be equal to the number of dimensions (or rank) of the template specified by *template-ref* or of the node array specified by *node-ref*.

- The number of *margin-width*, if any, must be equal to the number of dimensions (or rank) of the template specified by *template-ref* or of the node array specified by *node-ref*.
- *reduction-spec* must have one or more *location-variable*'s if and only if *reduction-kind* is either `FIRSTMAX`, `FIRSTMIN`, `LASTMAX`, or `LASTMIN`.

Examples

Example 1

XcalableMP Fortran

```
!$xmp distribute t(block) onto p
!$xmp align (i) with t(i) :: a, b
      ...
!$xmp loop (i) on t(i)
      do i = 1, N
          a(i) = 1.0
          b(i) = a(i)
      end do
```

The `loop` construct determines the node that executes each of the iterations, according to the distribution of template `t`, and distributes the execution. This example is syntactically equivalent to the one shown below, but will be faster because the iterations to be executed by each node can be determined before executing the loop.

XcalableMP Fortran

```
!$xmp distribute t(block) onto p
!$xmp align (i) with t(i) :: a, b
      ...
      do i = 1, N
!$xmp task on t(i)
          a(i) = 1.0
          b(i) = a(i)
!$xmp end task
      end do
```

Example 2

XcalableMP Fortran

```
!$xmp distribute t(*,block) onto p
!$xmp align (i,j) with t(i,j) :: a, b
      ...
!$xmp loop (i,j) on t(i,j)
      do j = 1, M
          do i = 1, N
              a(i,j) = 1.0
              b(i,j) = a(i,j)
          end do
      end do
```

Because the first dimension of template `t` is not distributed, only the `j` loop, which is aligned with the second dimension of `t`, is distributed. This example is syntactically equivalent to the `task` construct shown below.

XcalableMP Fortran
```
!$xmp distribute t(*,block) onto p
!$xmp align (*,j) with t(*,j) :: a, b
      ...
      do j = 1, M
!$xmp task on t(*,j)
          do i = 1, N
              a(i,j) = 1.0
              b(i,j) = a(i,j)
          end do
!$xmp end task
      end do
```

Example 3

XcalableMP Fortran
```
!$xmp distribute t(block,block) onto p
!$xmp align (i,j) with t(i,j) :: a, b
      ...
!$xmp loop (i,j) on t(i,j)
      do j = 1, M
          do i = 1, N
              a(i,j) = 1.0
              b(i,j) = a(i,j)
          end do
      end do
```

The distribution of loops in the nested loop can be specified using the sequence of *loop-indexes* in one `loop` construct. This example is equivalent to the loop shown below, but will run faster because the iterations to be executed by each node can be determined outside of the nested loop. Note that the node set specified by the inner `on` clause is a subset of that specified by the outer one.

XcalableMP Fortran
```
!$xmp distribute t(block,block) onto p
!$xmp align (i,j) with t(i,j) :: a, b
      ...
!$xmp loop (j) on t(:,j)
      do j = 1, M
!$xmp loop (i) on t(i,j)
          do i = 1, N
              a(i,j) = 1.0
              b(i,j) = a(i,j)
          end do
      end do
```

Example 4

XcalableMP Fortran
```
!$xmp nodes p(10,3)
      ...
!$xmp loop on p(:,i)
      do i = 1, 3
```

```
          call subtask ( i )
      end do
```

Three node sets `p(:,1)`, `p(:,2)`, and `p(:,3)` are created as the executing node sets, and each of them executes iterations `1`, `2`, and `3` of the associated loop, respectively. This example is equivalent to the loop containing `task` constructs (below left) or static `tasks/task` constructs (below right).

XcalableMP Fortran

```
!$xmp nodes p(10,3)
      ...
      do i = 1, 3
!$xmp task on p(:,i)
          call subtask ( i )
!$xmp end task
      end do
```

XcalableMP Fortran

```
!$xmp nodes p(10,3)
      ...
!$xmp tasks
!$xmp task on p(:,1)
      call subtask ( 1 )
!$xmp end task
!$xmp task on p(:,2)
      call subtask ( 2 )
!$xmp end task
!$xmp task on p(:,3)
      call subtask ( 3 )
!$xmp end task
!$xmp end tasks
```

Example 5

XcalableMP Fortran

```
      ...
      lb(1)  = 1
      iub(1) = 10
      lb(2)  = 11
      iub(2) = 25
      lb(3)  = 26
      iub(3) = 50
!$xmp loop (i) on p(lb(i):iub(i))
      do i = 1, 3
          call subtask ( i )
      end do
```

The executing node sets of different sizes are created by `p(lb(i):iub(i))` with different values of i for unbalanced workloads. This example is equivalent to the loop containing `task` constructs (below left) or static `tasks/task` constructs (below right).

XcalableMP Fortran

```
      do i = 1, 3
!$xmp task on p(lb(i):iub(i))
          call subtask ( i )
!$xmp end task
      end do
      ...
```

XcalableMP Fortran

```
!$xmp tasks
!$xmp task on p(1:10)
      call subtask ( 1 )
!$xmp end task
!$xmp task on p(11:25)
      call subtask ( 2 )
!$xmp end task
!$xmp task on p(25:50)
      call subtask ( 3 )
!$xmp end task
!$xmp end tasks
```

Example 6

XcalableMP Fortran

```
      ...
      s = 0.0
!$xmp loop (i) on t(i) reduction(+:s)
      do i = 1, N
          s = s + a(i)
      end do
```

This loop computes the sum of `a(i)` into the variable `s` on each node. Note that only the partial sum is computed on `s` without the reduction clause. This example is equivalent to the code given below.

XcalableMP Fortran

```
      ...
      s = 0.0
!$xmp loop (i) on t(i)
      do i = 1, N
          s = s + a(i)
      end do
!$xmp reduction(+:s) on t(1:N)
```

Example 7

XcalableMP Fortran

```
      ...
      amax = -1.0e30
      ip = -1
      jp = -1
!$xmp loop (i,j) on t(i,j) reduction(firstmax:amax/ip,jp/)
      do j = 1, M
          do i = 1, N
              if( 1(i,j) .gt. amx ) then
                  amx = a(i,j)
                  ip = i
                  jp = j
              end if
```

```
        end do
    end do
```

This loop computes the maximum value of `a(i,j)` and stores it into the variable `amax` in each node. In addition, the first indices for the maximum element of `a` are obtained in `ip` and `jp` after executing the loops. Note that this example cannot be written using the `reduction` construct.

Example 8

XcalableMP Fortran

```
!$xmp loop (i,j) on t(i,j) expand(/unbound/1,/unbound/1)
      do j = 1, 16
          do i = 1, 16
              ...
          end do
      end do

!$xmp loop (i,j) on t(i,j) margin(/unbound/1,/unbound/1)
      do j = 1, 16
          do i = 1, 16
              ...
          end do
      end do
```

Assuming that the template `t(100,100)` is distributed in (block,block) onto a node array `p(4,4)`, the original local iteration space on `p(1,1)`, $orig_{1,1}$ is:

$$orig_{1,1} = \{ \begin{array}{cccc} (1,1), & (2,1), & (3,1), & (4,1), \\ (1,2), & (2,2), & (3,2), & (4,2), \\ (1,3), & (2,3), & (3,3), & (4,3), \\ (1,4), & (2,4), & (3,4), & (4,4) \end{array} \}$$

and it is expanded using the `expand` clause for the first loop, as follows:

$$expand(1,1)_{1,1} = \{ \begin{array}{cccccc} (0,0), & (0,1), & (0,2), & (0,3), & (0,4), & (0,5), \\ (1,0), & (1,1), & (1,2), & (1,3), & (1,4), & (1,5), \\ (2,0), & (2,1), & (2,2), & (2,3), & (2,4), & (2,5), \\ (3,0), & (3,1), & (3,2), & (3,3), & (3,4), & (3,5), \\ (4,0), & (4,1), & (4,2), & (4,3), & (4,4), & (4,5), \\ (5,0), & (5,1), & (5,2), & (5,3), & (5,4), & (5,5) \end{array} \}$$

Note that $expand(1,1)_{1,1}$ spreads out of the original global iteration space $\{(i,j) \,|\, 1 \leq i,j \leq 16\}$ because the `/unbound/` specifier is specified in the `expand` clause.

The local iteration space for the second loop with the `margin` clause is defined using the symmetric difference operator, as follows:

$$\begin{array}{rcl} margin(1,1)_{1,1} & = & expand(1,1)_{1,1} \triangle orig_{1,1} \\ & = & \{ \begin{array}{cccccc} (0,0), & (0,1), & (0,2), & (0,3), & (0,4), & (0,5), \\ (1,0), & & & & & (1,5), \\ (2,0), & & & & & (2,5), \\ (3,0), & & & & & (3,5), \\ (4,0), & & & & & (4,5), \\ (5,0), & (5,1), & (5,2), & (5,3), & (5,4), & (5,5) \end{array} \} \end{array}$$

4.4.4 `array` Construct

Synopsis

The `array` construct divides the work of an array assignment between nodes.

Syntax

[F] `!$xmp array on` *template-ref*
array-assignment-statement

[C] `#pragma xmp array on` *template-ref*
array-assignment-statement

Description

The array assignment is an alternative to a loop that performs an assignment to each element of an array. This directive specifies the parallel execution of an array assignment, where each sub-assignment and sub-operation of an element is executed by a node that is determined by the `on` clause.

Note that array assignments can also be used in XcalableMP C, which is one of the language extensions introduced by XcalableMP (see Section 3.2).

Restrictions

- The node set specified by *template-ref* in the `on` clause must be a subset of the parent node set.
- The template section specified by *template-ref* must have the same shape as the associated array assignment.
- The `array` construct is global and must be executed by all of the executing nodes with the same valuse for the variables that appear in the construct.

Examples

Example 1

XcalableMP Fortran

```
!$xmp distribute t(block) onto p
!$xmp align (i) with t(i) :: a
      ...
!$xmp array on t(1:N)
      a(1:N) = 1.0
```

This example is equivalent to the code shown below.

XcalableMP Fortran

```
!$xmp distribute t(block) onto p
!$xmp align (i) with t(i) :: a
      ...
!$xmp loop on t(1:N)
      do i = 1, N
          a(i) = 1.0
      end do
```

Example 2

XcalableMP Fortran

```
!$xmp template t(100,20)
!$xmp distribute t(block,block) onto p
      dimension a(100,20), b(100,20)
!$xmp align (i,j) with t(i,j) :: a, b
      ...
!$xmp array on t
      a = b + 2.0
```

This example is equivalent to the code shown below.

XcalableMP Fortran

```
!$xmp template t(100,20)
!$xmp distribute t(block,block) onto p
      dimension a(100,20), b(100,20)
!$xmp align (i,j) with t(i,j) :: a, b
      ...
!$xmp loop (i,j) on t(i,j)
      do j = 1, 20
         do i = 1, 100
            a(i,j) = b(i,j) + 2.0
         end do
      end do
```

4.5 Global-view Communication and Synchronization Constructs

4.5.1 `reflect` Construct

Synopsis

The `reflect` construct assigns the value of a reflection source to the corresponding shadow object.

Syntax

[F] `!$xmp reflect (` *array-name [, array-name]...* `)` ▉
▉ *[*`width (` *reflect-width [, reflect-width]...* `)` *] [*`orthogonal`*] [*`async (` *async-id* `)` *]*

[C] `#pragma xmp reflect (` *array-name [, array-name]...* `)` ▉
▉ *[*`width (` *reflect-width [, reflect-width]...* `)` *] [*`orthogonal`*] [*`async (` *async-id* `)` *]*

where *reflect-width* must be one of:

[`/periodic/`*] int-expr*
[`/periodic/`*] int-expr* : *int-expr*

Description

The `reflect` construct updates each of the shadow objects of the array specified by *array-name* with the value of its corresponding reflection source. Note that the shadow objects corresponding to elements at the non-orthogonal positions are also updated with this construct, unless the `orthogonal` clause is specified.

When the `width` clause is specified and takes the form "*int-expr* : *int-expr*" in a dimension, the shadow area having the width specified by the first *int-expr* at the lower bound and that specified by the second one at the upper bound in the dimension are updated. When the `width` clause is specified, and takes the form *int-expr*, the shadow areas having the same width specified at both the upper and lower bounds in the dimension are updated. When the `width` clause is omitted, the whole shadow area of the array is updated.

In particular, when the `/periodic/` modifier is specified in *reflect-width*, the update of the shadow object in the dimension is "periodic," which means that the shadow object at the global lower (upper) bound is treated as if it corresponds to the data object of the global upper (lower) bound, and is updated with that value by the `reflect` construct.

When the `orthogonal` clause is specified, only the shadow objects corresponding to elements at the orthogonal positions are updated by the `reflect` construct.

When the `async` clause is specified, the statements following this construct may be executed before the operation is complete.

Restrictions

- The arrays specified by the sequence of *array-names* must be mapped onto the executing node set.
- The reflect width of each dimension specified by the *reflect-width* must not exceed the shadow width of the arrays.
- The `reflect` construct is global, which means that it must be executed by all nodes in the current executing node set, and each local variable referenced in the construct must have the same value among all of them.
- *async-id* must be an expression of type default integer in XcalableMP Fortran or type `int` in XcalableMP C.

Example

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(100)
!$xmp distribute t(block) onto p

      real a(100)
!$xmp align a(i) with t(i)
!$xmp shadow a(1)

      ...
!$xmp reflect (a) width (/periodic/1)
```

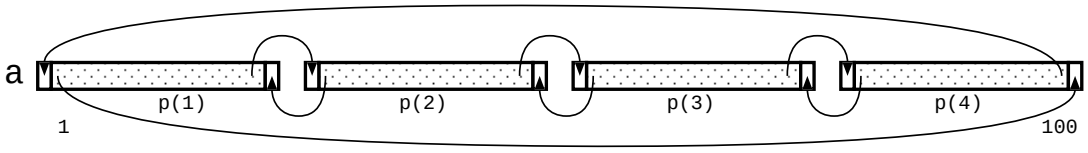

Figure 4.2: Example of periodic shadow reflection.

The `shadow` directive allocates "periodic" shadow areas of the array `a`. The `reflect` construct updates "periodically" the shadow area of `a` (Figure 4.2). A periodic shadow at the lower bound on the node `p(1)` is updated with the value of `a(100)` and that at the upper bound on `p(4)` with the value of `a(1)`.

4.5.2 `gmove` Construct

Synopsis

The `gmove` construct allows an assignment statement, which may cause communication, to be executed possibly in parallel by the executing nodes.

Syntax

```
[F]  !$xmp gmove [in | out] [async ( async-id )]
[C]  #pragma xmp gmove [in | out] [async ( async-id )]
```

Description

This construct copies the value of the right-hand side variable into the left-hand side of the associated assignment statement, which may cause communication between the executing nodes. Such communication is detected, scheduled, and performed by the XcalableMP runtime system.

There are three operating modes of the `gmove` construct:

- **collective mode**

  When neither the `in` nor the `out` clause is specified, the copy operation is performed collectively, and results in implicit synchronization among the executing nodes.

  If the `async` clause is not specified, then the construct is "synchronous," and it is guaranteed that the left-hand side data can be read and overwritten, the right-hand side data can be overwritten, and all of the operations of the construct on the executing nodes are completed when returning from the construct; otherwise, the construct is "asynchronous," and it is not guaranteed that the operations are completed, until the associating `wait_async` construct (Section 4.5.6) is completed.

- **in mode**

  When the `in` clause is specified, the right-hand side data of the assignment, all or part of which may reside outside the executing node set, can be transferred from its owner nodes to the executing nodes by this construct.

  If the `async` clause is not specified, then the construct is "synchronous," and it is guaranteed that the left-hand side data can be read and overwritten, and that all of the operations of the construct on the owner nodes of the right-hand side and the executing nodes are completed when returning from the construct; otherwise, the construct is "asynchronous," and it is not guaranteed that the operations are completed, until the associating `wait_async` construct (Section 4.5.6) is completed.

- **out mode**

  When the `out` clause is specified, the left-hand side data of the assignment, all or part of which may reside outside the executing node set, can be transferred from the executing nodes to its owner nodes by this construct.

  If the `async` clause is not specified, then the construct is "synchronous," and it is guaranteed that the right-hand side data can be overwritten, and that all of the operations of

the construct on the owner nodes of the left-hand side and the executing nodes are completed when returning from the construct; otherwise, the construct is "asynchronous," and it is not guaranteed that the operations are completed, until the associating `wait_async` construct (Section 4.5.6) is completed.

When the `async` clause is specified, the statements following this construct may be executed before the operation is complete.

Restrictions

- The `gmove` construct must be followed by (i.e., associated with) a simple assignment statement that contains neither arithmetic operations nor function calls.
- The `gmove` construct is global, which means that it must be executed by all nodes in the current executing node set, and each local variable referenced in the construct must have the same value.
- If the `gmove` construct is in the *collective* mode, then all elements of the distributed arrays appearing on both the left-hand side and the right-hand side of the associated assignment statement must reside in the executing node set.
- If the `gmove` construct is in the *in* mode, then all elements of the distributed array appearing on the left-hand side of the associated assignment statement must reside in the executing node set.
- If the `gmove` construct is in the *out* mode, then all elements of the distributed array appearing on the right-hand side of the associated assignment statement must reside in the executing node set.
- *async-id* must be an expression of type default integer in XcalableMP Fortran or type `int` in XcalableMP C.

Examples

Example 1: Array assignment If the arrays on both the left-hand side and the right-hand side are distributed, then the copy operation is performed using all-to-all communication. If the left-hand side is a replicated array, this copy is performed using multi-cast communication. If the right-hand side is a replicated array, then no communication is required.

XcalableMP Fortran

```
!$xmp gmove
      a(:,1:N) = b(:,3,0:N-1)
```

XcalableMP C

```
#pragma xmp gmove
      a[1:N][:] = b[0:N][3][:];
```

Example 2: Scalar assignment to an array When the right-hand side is an element of a distributed array, the copy operation is performed by broadcast communication from the owner of the element. If the right-hand side is a replicated array, then no communication is required.

XcalableMP Fortran

```
!$xmp gmove
      a(:,1:N) = c(k)
```

XcalableMP C

```
#pragma xmp gmove
      a[1:N][:] = c[k]
```

Example 3: in mode assignment Because `b(3)` referenced on the right-hand side of the `gmove` construct does not reside in the executing node set (`p(1:2)`), the construct is executed in the *in* mode. Thus, `b(3)` is transferred from its owner node `p(3)` to the executing node set.

Until `p(1:2)` returns from the construct, there is no gurantee that any node can read and overwrite `a(1:2)`, and that any relevant operations on `p(1:2)` and `p(3)` are completed.

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(4)
!$xmp distribute t(block) onto p

      real a(4), b(4)
!$xmp align (i) with t(i) : a, b
      ...
!$xmp task on p(1:2)
      ...
!$xmp gmove in
      a(1:2) = b(2:3)
      ...
!$xmp end task
```

4.5.3 `barrier` Construct

Synopsis

The `barrier` construct specifies an explicit barrier at the point at which the construct appears.

Syntax

[F] `!$xmp barrier` *[*`on` *nodes-ref |template-ref]*
[C] `#pragma xmp barrier` *[*`on` *nodes-ref | template-ref]*

Description

The barrier operation is performed among the node set specified by the `on` clause. If no `on` clause is specified, then it is assumed that the current executing node set is specified in it.

Note that an `on` clause may represent multiple node sets. In such a case, a barrier operation is performed in each node set.

Restriction

- The node set specified by the `on` clause must be a subset of the executing node set.

4.5.4 `reduction` Construct

Synopsis

The `reduction` construct performs a reduction operation among nodes.

Syntax

[F] `!$xmp reduction (` *reduction-kind : variable [, variable]...* `)` ▉
▉ *[*`on` *node-ref | template-ref] [*`async (` *async-id* `)`*]*

where *reduction-kind* is one of:

```
+
*
.and.
.or.
.eqv.
.neqv.
max
min
iand
ior
ieor
```

[C] `#pragma xmp reduction (` *reduction-kind : variable [, variable]...* `)` ▉
▉ *[*`on` *node-ref | template-ref] [*`async (` *async-id* `)`*]*

where *reduction-kind* is one of:

```
+
*
&
|
^
&&
||
max
min
```

Description

The `reduction` construct performs a type of reduction operation specified by *reduction-kind* for the specified local variables among the node set specified by the `on` clause, and it sets the reduction results to the variables on each of the nodes. Note that some of the reduction operations, namely, `FIRSTMAX`, `FIRSTMIN`, `LASTMAX`, and `LASTMIN`, which can be specified in the `reduction` clause of the `loop` directive, cannot be specified in the `reduction` construct because their semantics are not defined for it. The variable specified by *variable*, which is the target of the reduction operation, is referred to as the "reduction variable." After the reduction operation, the value of a reduction variable becomes the same in every node that performs the operation.

The reduction result is computed by combining the reduction variables on all of the nodes using the reduction operator. The ordering of this reduction is unspecified.

When the `async` clause is specified, the statements following this construct may be executed before the operation is complete.

When *template-ref* is specified in the `on` clause, the operation is performed in a node set that consists of nodes onto which the specified template section is distributed. Therefore, before the `reduction` construct is executed, the referenced template must be fixed. When *nodes-ref* is specified in the `on` clause, the operation is performed in the specified node set. When the `on` clause is omitted, the operation is performed in the executing node set.

Note that an `on` clause may represent multiple node sets. In such a case, a reduction operation is performed in each node set.

Restrictions

- The variables specified by the sequence of *variables* must either not be aligned or must be replicated among nodes of the node set specified by the `on` clause.
- The `reduction` construct is global, which means that it must be executed by all nodes in the current executing node set, and each local variable referenced in the construct must have the same value.
- *async-id* must be an expression of type default integer in XcalableMP Fortran or type `int` in XcalableMP C.
- The node set specified by the `on` clause must be a subset of the executing node set.

Examples

Example 1

XcalableMP Fortran

```
!$xmp reduction(+:s)
!$xmp reduction(max:aa) on t(*,:)
!$xmp reduction(min:bb) on p(10:30)
```

In the first line, the reduction operation calculates the sum of the scalar variable `s` in the executing node set, and the result is stored in the variable in each node.

The reduction operation in the second line computes the maximum value of the variable `aa` in each node set onto which each of the template sections specified by `t(*,:)` is distributed.

In the third line, the minimum value of the variable `bb` in the node set specified by `p(10:30)` is calculated. This example is equivalent to the following code using the `task` construct.

XcalableMP Fortran

```
!$xmp task on p(10:30)
!$xmp reduction(min:bb)
!$xmp end task
```

Example 2

XcalableMP Fortran

```
      dimension a(n,n), p(n), w(n)
!$xmp align a(i,j) with t(i,j)
!$xmp align p(i) with t(i,*)
!$xmp align w(j) with t(*,j)
      ...
!$xmp loop (j) on t(:,j)
      do j = 1, n
          sum = 0
!$xmp loop (i) on t(i,j) reduction(+:sum)
          do i = 1, n
              sum = sum + a(i,j) * p(i)
          end do
```

```
          w(j) = sum
      end do
```

This code computes the matrix vector product, where a `reduction` clause is specified for the `loop` construct of the inner loop. This is equivalent to the following code snippet.

XcalableMP Fortran

```
!$xmp loop (j) on t(:,j)
      do j = 1, n
          sum = 0
!$xmp loop (i) on t(i,j)
          do i = 1, n
              sum = sum + a(i,j) * p(i)
          end do
!$xmp reduction(+:sum) on t(1:n,j)
          w(j) = sum
      end do
```

In these cases, the reduction operation on the scalar variable `sum` is performed for every iteration in the outer loop, which may cause a large overhead. To reduce this overhead, the `reduction` clause should be specified in the `loop` construct for the outer loop. This is because the node set in which the reduction operation is performed is determined on the basis of its `on` clause (see 4.4.3), and the `on` clause of the outer `loop` construct is different from that of the inner one. However, this code can be modified using the `reduction` construct as follows:

XcalableMP Fortran

```
      dimension a(n,n), p(n), w(n)
!$xmp align a(i,j) with t(i,j)
!$xmp align p(i) with t(i,*)
!$xmp align w(j) with t(*,j)
      ...
!$xmp loop (j) on t(:,j)
      do j = 1, n
          sum = 0
!$xmp loop (i) on t(i,j)
          do i = 1, n
              sum = sum + a(i,j) * p(i)
          end do
          w(j) = sum
      end do
!$xmp reduction(+:w) on t(1:n,*)
```

This code performs a reduction operation on the array `w` only once, which may result in faster operation.

4.5.5 `bcast` Construct

Synopsis

The `bcast` construct performs broadcast communication from a specified node.

Syntax

[F] `!$xmp bcast (` *variable [, variable]...* `)` *[*`from` *nodes-ref | template-ref]* ▉
▉ *[*`on` *nodes-ref] | template-ref] [*`async (` *async-id* `)]`

[C] `#pragma xmp bcast (` *variable [, variable]...* `)` *[*`from` *nodes-ref | template-ref]* ▉
▉ *[*`on` *nodes-ref | template-ref] [*`async (` *async-id* `)]`

Description

The values of the variables specified by the sequence of *variables* (called *broadcast variables*) are broadcasted from the node specified by the `from` clause (called the *source node*) to each of the nodes in the node set specified by the `on` clause. After executing this construct, the values of the broadcast variables become the same as those in the source node. If the `from` clause is omitted, then the *first* node, that is, the leading one in Fortran's array element order, of the node set specified by the `on` clause is assumed to be a source node. If the `on` clause is omitted, then it is assumed that the current executing node set is specified in it.

When the `async` clause is specified, the statements following this construct may be executed before the operation is complete.

Restrictions

- The variables specified by the sequence of *variables* must either not be aligned or must be replicated among nodes of the node set specified by the `on` clause.
- The `bcast` construct is global, which means that it must be executed by all nodes in the current executing node set, and each local variable referenced in the construct must have the same value among all of them.
- *async-id* must be an expression of type default integer in XcalableMP Fortran or type `int` in XcalableMP C.
- The node set specified by the `on` clause must be a subset of the executing node set.
- The source node specified by the `from` clause must belong to the node set specified by the `on` clause.
- The source node specified by the `from` clause must be one node.

4.5.6 `wait_async` Construct

Synopsis

The `wait_async` construct guarantees that asynchronous communications specified by *async-id* are complete.

Syntax

[F] `!$xmp wait_async (` *async-id [, async-id]...* `)` *[*`on` *nodes-ref | template-ref]*

[C] `#pragma xmp wait_async (` *async-id [, async-id]...* `)` *[*`on` *nodes-ref | template-ref]*

Description

The `wait_async` construct will block, and therefore statements following it will not be executed, until the completion of all of the asynchronous communications that are specified by *async-id*'s and issued on the node set specified by the `on` clause. If an *async-id* is not associated with any asynchronous communication, the `wait_async` construct ignores it.

Restrictions

- *async-id* must be an expression of type default integer in XcalableMP Fortran or type `int` in XcalableMP C.
- ~~*async-id* must be associated with an asynchronous communication using the `async` clause of a communication construct.~~
- The `wait_async` construct is global, which means that it must be executed by all nodes in the current executing node set, and each local variable referenced in the construct must have the same value among all of them.
- The node set specified by the `on` clause must be the same as those of the global constructs that initiate the asynchronous communications specified by *async-id*.

4.5.7 `async` Clause

Synopsis

The `async` clause of the `reflect`, `gmove`, `reduction`, and `bcast` constructs enables the corresponding communication to be performed asynchronously.

Description

Communication corresponding to the construct with an `async` clause is performed asynchronously, that is, it is initiated but not completed, and therefore, statements following it may be executed before the communication is complete.

Example

XcalableMP Fortran

```
!$xmp reflect (a) async(1)
      S1
!$xmp wait_async(1)
      S2
```

The `reflect` construct on the first line matches the `wait` construct on the third line because both of their *async_id* evaluate to one. These constructs ensure that statements in `S1` can be executed before the `reflect` communication is complete, and no statement in `S2` is executed until the `reflect` communication is complete.

4.5.8 `reduce_shadow` Construct

Synopsis

The `reduce_shadow` construct adds values of shadow objects to their reflection source.

Syntax

```
[F]  !$xmp reduce_shadow ( array-name [, array-name]... ) &
       & [width ( reflect-width [, reflect-width]... )] [orthogonal] [async ( async-id )]
[C]  #pragma xmp reduce_shadow ( array-name [, array-name]... ) &
       & [width ( reflect-width [, reflect-width]... )] [orthogonal] [async ( async-id )]
```

Description

The `reduce_shadow` construct adds values of shadow objects of the array specified by *array-name* to their reflection source. Note that the shadow objects corresponding to elements at the non-orthogonal positions are also added as the default behavior.

When the `width` clause is specified and has the form "*int-expr* : *int-expr*" in a dimension, the shadow areas having the width specified by the first *int-expr* at the lower bound, and that specified by the second one at the upper bound in the dimension are added. When the `width` clause is specified and has the form *int-expr*, the shadow areas having the same width specified at both the upper and lower bounds in the dimension are added. When the `width` clause is omitted, the whole shadow area of the array is added.

In particular, when the `/periodic/` modifier is specified in *reflect-width*, the addition of the shadow object in the dimension is "periodic," which means that the shadow object at the global lower (upper) bound is treated as if it corresponds to the data object of the global upper (lower) bound and is added by the `reduce_shadow` construct.

When the `orthogonal` clause is specified, the shadow object is added only by orthogonal nodes.

When the `async` clause is specified, the statements following this construct may be executed before the operation is complete.

Restrictions

- The arrays specified by the sequence of *array-names* must be mapped onto the executing node set.
- The width of each dimension specified by *reflect-width* must not exceed the shadow width of the arrays.
- The `reduce_shadow` construct is global, which means that it must be executed by all nodes in the current executing node set, and each local variable referenced in the construct must have the same value among all of them.
- *async-id* must be an expression of type default integer in XcalableMP Fortran or type `int` in XcalableMP C.

Examples

XcalableMP Fortran

```
      real rho(n,n)
!$xmp align rho(i,j) with t1(i,j)
!$xmp shadow rho(1:1)

      real f(m)
      integer x(m), y(m)
!$xmp align (k) with t2(k) : f, x, y

!$xmp loop on t2(k)
      do i = 1, no
        ix = x(i)
        iy = y(i)
        dx = x(i) - ix
        dy = y(i) - iy
        rho(ix  ,iy  ) = rho(ix  ,iy  ) + (1.0-dx)*(1.0-dy)*f(i)
```

```
        rho(ix+1,iy  ) = rho(ix+1,iy  ) +      dx *(1.0-dy)*f(i)
        rho(ix  ,iy+1) = rho(ix  ,iy+1) + (1.0-dx)*     dy *f(i)
        rho(ix+1,iy+1) = rho(ix+1,iy+1) +      dx *     dy *f(i)
      end do

!$xmp reduce_shadow (rho)
!$xmp reflect (rho)
```

Assume that a two-dimensional field `rho` and m particles are both distributed onto nodes. On each node, a contribution of a particle `f(k)` is added to the nearest grid point of the field and its neighbors, which may be in the shadow area on the node. In the last two lines, the values of the shadow area from neighboring nodes are added to the corresponding data object, and the results are then copied back to the shadow area on the neighboring nodes.

Chapter 5

Support for the Local-view Programming

This chapter describes the coarray features in XcalableMP, which are based on that of Fortran 2008. Note that they are also available in XcalableMP C. In addition, this chapter describes some directives for the local-view programming.

The coarray features in Fortran 2008 are extended and integrated into XcalableMP. The specifications in this chapter are designed to achieve the following purposes:

- Upward compatibility to the Fortran 2008 coarray features
  If an XcalableMP Fortran program does not contain any XMP directives, any standard-conforming Fortran 2008 program remains standard conforming under XcalableMP. In this sense, the interpretations and extensions defined in this chapter are upward compatible with the Fortran International Standard, ISO/IEC 1539-1:2010 (Fortran 2008).

- Support for task parallelism
  XcalableMP makes it possible to construct a task parallel program by combining multiple Fortran 2008 codes, which may be developed independently, with minimum modifications.

- Integration of global-view style programming and local-view style programming
  In XcalableMP, users can appropriately use global-view style programming of XcalableMP or local-view style programming, which is typically used in MPI or Fortran 2008 programs, according to the characteristics of the code in a program.

- Possibility of support for multiple topologies of a computing system
  An XcalableMP processor may allow users to specify the correspondence between node arrays and the topologies of a computing system, and to exploit the full potential of a particular system.

5.1 Rules Determining Image Index

This section defines how the image index of an image in a set of images is determined in association with a node array and a `task` construct.

5.1.1 Primary Image Index

Every image has a default image index in all of the images at the invocation of a program. In XcalableMP, the default image index is the primary image index, and is an integer value that ranges from one to the number of images at the invocation of a program.

A primary node array corresponds to all of the images at the invocation of a program, and it also corresponds to all of the nodes at the invocation of a program. The primary image index of an image is the (Fortran) subscript order value of the uniquely corresponding element of a primary node array.

5.1.2 Image Index Determined by a `task` Directive

The execution of a `task` directive determines that a set of nodes (and the corresponding set of images) forms an executing node set. If a name of a node array or a subobject of a node array appears in the `task` directive, the nodes and the corresponding images in the executing node set are ordered in (Fortran) array element order in the node array or the subobject of the node array. If a name of a template array or a subobject of a template array appears in the `task` directive, the nodes and the corresponding images in the executing node set are ordered in (Fortran) array element order in the corresponding subobject of the node array. The image index of an image in the determined set of images is the integer order value in the range one to the cardinality of the set of images.

5.1.3 Current Image Index

The image index of an image in the current set of images is the current image index.

A current executing node array corresponds to the current set of images and also the current executing node set at the evaluation of the declaration of the node array. Each image in the current set of images corresponds to the element of an executing node array whose subscript order value is the same as the current image index of the image when the evaluation of the declaration of the executing node array is being evaluated. In particular, when all `task` directive constructs are completed, the current image index of an image is the same as the primary image index.

5.1.4 Image Index Determined by a Non-primary Node Array

A non-primary node array corresponds to all of the images at the invocation of a program, and it also corresponds to all the nodes at the invocation of a program. The correspondence between each image and each element of a non-primary node array is processor dependent. A processor may support any means to specify the correspondence.

The image index of an image in all of the images at the invocation of a program is the subscript order value of the corresponding element of a non-primary node array. This is the case if and only if the current set of images corresponds to the non-primary node whole array in which the nodes in the executing node set are ordered in (Fortran) array-element order in the non-primary node whole array. The image index is a non-primary image index.

The correspondence between the primary image index and a non-primary image index of the same image is processor dependent. Between any two distinct non-primary node arrays, the correspondence between a non-primary image index and the other non-primary image index of the same image is processor dependent unless they have the same shape. If two non-primary node arrays have the same shape, the corresponding elements of the node arrays correspond to the same image.

5.1.5 Image Index Determined by an Equivalenced Node Array

A `nodes` directive with "=*node-ref*" that is not "=*" or "=**" specifies that each element of the declared node array corresponds in (Fortran) array-element order to that of the *node-ref*, which is the name of a node array or a subobject of a node array. The nodes in the declared node array and the corresponding images are ordered in (Fortran) array-element order in the

node-ref. The image index of an image in the set of images corresponding to the declared node array is the integer order value ranging from one to the cardinality of the set of images.

5.1.6 On-node Image Index

XcalableMP supports the `coarray` directive and the `image` directive to specify that an image index indicates the image corresponding to the element of a particular node array whose subscript order value is the same as the image index. The image index is an on-node image index for the specified node array. Because an evaluation of the declaration of a node array determines a set of images corresponding to the node array, the directives specify that the set of images is the "all images" for the image indices affected by the directives. In particular, the on-node image index for a primary node array is the primary image index.

5.2 Basic Concepts

In XcalableMP, "all images" in Fortran 2008 changes coupled with the execution of `task` constructs, and refers to the current set of images. In particular, when an `allocate` statement is executed for which an *allocate-object* is a coarray, there is an implicit synchronization of all the images in the current set of images. On each image in the current set of images, execution of the segment following the statement is delayed until all other images in the set have executed the same statement the same number of times. When a `deallocate` statement is executed for which an *allocate-object* is a coarray, there is an implicit synchronization of all the images in the current set of images. On each image in the current set of images, execution of the segment following the statement is delayed until all other images in the set have executed the same statement the same number of times.

- When an allocatable coarray is allocated during the execution of `task` constructs, the coarray shall be subsequently deallocated before the completion of the `task` construct whose `task` directive is the most recently executed one in the `task` constructs that are not completed at the allocation.

The image index determined by an image selector indicates the current image index by default. Coarrays are visible within the range of the "all images," and are accessed using the current image index by default. The image index that appears in an executable statement indicates the current image index by default.

5.2.1 Examples

- In the following code fragment, the value of a coarray `b` on the images 1, 2, 3, and 4, which constitute the executing node set and correspond to `node(5)`, `node(6)`, `node(7)`, and `node(8)` respectively, is defined with the value of the coarray `a` on `node(5)`.

XcalableMP Fortran

```
      program xmpcoarray
!$xmp nodes node(8)=**   ! A primary node array.
!$xmp task on node(5:8)  ! The executing node set
        call sub         ! corresponds to node(5:8).
!$xmp end task
      end

      subroutine sub
      real, save :: a[*], b[*] ! The images 1, 2, 3,
```

```
        :                       ! and 4 correspond to node(5:8),
      b = a[1]                  ! respectively.
```

- In the following code fragment, an allocatable coarray `a` is allocated on the images 1, 2, 3, and 4, which constitute the executing node set and correspond to `node(5)`, `node(6)`, `node(7)`, and `node(8)`, respectively.

XcalableMP Fortran

```
      program xmpcoarray
!$xmp nodes node(8)=**
!$xmp task on node(5:8)  ! The executing node set
        call sub2        ! corresponds to node(5:8).
!$xmp end task
      end

      subroutine sub2
      real, allocatable :: a(:)[:]
        :
      allocate(a(0:99)[*])
```

Note

- The result of `xmp_num_nodes()` is always the same as that of `NUM_IMAGES()`.
- The result of `xmp_node_num()` is always the same as that of `THIS_IMAGE()`.
- In a `read` statement, an io-unit that is an asterisk identifies an external unit that is preconnected for a sequential formatted input only on the image whose primary image index is one.

5.3 `coarray` Directive

5.3.1 Purpose and Form of the `coarray` Directive

The `coarray` directive maps coarrays onto a node array and the set of images that corresponds to the node array. An image index determined by an image selector for a coarray that appears in a `coarray` directive always indicates the on-node image index for the node array; that is, the specified image corresponds to the node whose subscript order value in the node array is the same as the image index.

A coarray appearing in a `coarray` directive is an on-node coarray of the node array that is specified in the `coarray` directive.

[F] `!$xmp coarray on` *node-name* `::` *object-name-list*

[C] `#pragma xmp coarray on` *node-name* `::` *object-name-list*

- An *object-name* shall be a name of a coarray declared in the same scoping unit.
- The same *object-name* shall not appear more than once in `coarray` directives in a scoping unit.

- If an *object-name* is a name of an allocatable object, the current set of images at the allocation and the deallocation of the object shall correspond to the node array specified as the *node-name*, and the current image index of each image shall be the same as the subscript order value of the corresponding element of the node array.

- If an *object-name* is the name of an allocated allocatable dummy argument, the set of images onto which it is mapped shall be a subset of the set of images that has most recently allocated the corresponding argument in the chain of argument associations.

- If an *object-name* is the name of a nonallocatable dummy argument whose ultimate argument has an allocatable attribute, the set of images onto which the *object-name* is mapped shall be a subset of the set of images that has most recently allocated the corresponding argument in the chain of argument associations.

- The image index determined by an image selector for an on-node coarray shall be within the range of one to the size of the node array onto which the on-node coarray is mapped.

- THIS_IMAGE(COARRAY[,DIM]) shall be invoked by the image contained in the set of images onto which the COARRAY argument is mapped if the COARRAY argument appears in a `coarray` directive.

Note

- The result value of THIS_IMAGE(COARRAY) is the sequence of cosubscript values for the COARRAY argument that would specify the current image index of the invoking image, if the COARRAY argument does not appear in a `coarray` directive. The result value of THIS_IMAGE(COARRAY) is the sequence of cosubscript values for the COARRAY argument that would specify the on-node image index of the invoking image for the node array onto which the COARRAY argument is mapped if the COARRAY argument appears in a `coarray` directive.

- The result value of THIS_IMAGE(COARRAY,DIM) is the value of cosubscript DIM in the sequence of cosubscript values for the COARRAY argument that would specify the current image index of the invoking image if the COARRAY argument does not appear in a `coarray` directive. The result value of THIS_IMAGE(COARRAY,DIM) is the value of cosubscript DIM in the sequence of cosubscript values for the COARRAY argument that would specify the on-node image index of the invoking image for the node array onto which the COARRAY argument is mapped if the COARRAY argument appears in a `coarray` directive.

5.3.2 An Example of the `coarray` Directive

XcalableMP Fortran

```
      module global
!$xmp nodes node(8)=**
      real s[*]              ! The coarray s is always
!$xmp coarray on node :: s   ! visible on node(1:8).
      end global

      program coarray
      use global
!$xmp task on node(5:8)      ! The executing node set
        call sub             ! consists of node(5:8).
```

```
!$xmp end task
      end

      subroutine sub
      use global
      real, save :: a[*]    ! The images 1, 2, 3, and 4
         :                  ! correspond to node(5:8), respectively.
      if(this_image().eq.1)then ! The value of the coarray a on node(5)
         s[1] = a               ! defines that of the coarray s on node(1)
      endif
```

5.4 image Directive

5.4.1 Purpose and Form of the image Directive

The image directive specifies that an image index in the following executable statement indicates the on-node image index of the node array specified in the image directive unless the image index is determined by an image selector.

The image directive also specifies that the execution of a sync all statement performs a synchronization of all of the images corresponding to the node array specified in the image directive.

[F] !$xmp image (*node-name*)

[C] #pragma xmp image (*node-name*)

- An image directive shall be followed by a sync all statement, an image control statement that contains *image-set*, or a reference to an intrinsic procedure that has IMAGES argument.

5.4.2 An Example of the image Directive

XcalableMP Fortran

```
      module global
!$xmp nodes node(8)=**
      real s[*]             ! The coarray s is always visible
!$xmp coarray on node :: s  ! on node(1:8).
      end global

      program image
      use global
!$xmp tasks
!$xmp task on node(1:4)
        call subA  ! The executing node set consists of node(1:4).
!$xmp end task
!$xmp task on node(5:8)
        call subB  ! The executing node set consists of node(5:8).
!$xmp end task
!$xmp end tasks
      end
```

```
      subroutine subA
      use global
      real, save :: a[*] ! The images 1, 2, 3, and 4
         :               ! correspond to node(1:4), respectively.
!$xmp image(node)        ! Synchronization between node(1:4) and
        sync images(5)   ! node(5).
      a = s[1]           ! a on node(1:4) is defined using
         :               ! the value of s on node(1).
      end subroutine

      subroutine subB
      use global
      real, save :: b[*] ! The images 1, 2, 3, and 4
         :               ! correspond to node(5:8), respectively.
      if(this_image() .eq. 1)then ! The image 1 indicates node(5).
        s[1] = b      ! s on node(1) is defined using the value of
                      ! b on node(5).
!$xmp   image(node)                ! Synchronization between
          sync images((/1,2,3,4/)) ! node(5) and node(1:4).
      endif
         :
      end subroutine
```

5.5 Image Index Translation Intrinsic Procedures

XcalableMP supports intrinsic procedures to translate image indices between different sets of images.

5.5.1 Translation to the Primary Image Index

xmp_get_primary_image_index(NUMBER,INDEX,PRI_INDEX,NODE_DESC)

Description. Translate image indices to the primary image indices.

Class. Subroutine.

Arguments.

NUMBER shall be a scalar of type default integer. It is an INTENT(IN) argument.

INDEX shall be a rank-one array of type default integer. The size of **INDEX** shall be greater than or equal to the value of **NUMBER**. It is an INTENT(IN) argument. The value of each element of **INDEX** shall be within the range one to the size of the node array specified in **NODE_DESC** if **NODE_DESC** appears. The value of each element of **INDEX** shall be within the range one to the cardinality of the current set of images if **NODE_DESC** does not appear.

PRI_INDEX shall be a rank-one array of type default integer. The size of **PRI_INDEX** shall be greater than or equal to the value of **NUMBER**. It is an INTENT(OUT) argument. If **NODE_DESC** appears, **PRI_INDEX(i)** is assigned the primary image index corresponding to the element of the node array specified in **NODE_DESC** whose subscript order value is **INDEX(i)**; otherwise, **PRI_INDEX(i)** is assigned

the primary image index corresponding to the image whose current image index is **INDEX(i)**.

NODE_DESC (optional) shall be a descriptor of a node array. It is an INTENT(IN) argument. **NODE_DESC** shall appear in XcalableMP C.

Example. In the following code fragment, the value of `index(1:4)` is `(/5,6,7,8/)`.

XcalableMP Fortran

```
!$xmp nodes node(1:8)=**        ! A primary node array
!$xmp nodes subnode(4)=node(5:8)
      integer index(4)
      call xmp_get_primary_image_index&
           &(4,(/1,2,3,4/),index,xmp_desc_of(subnode))
```

5.5.2 Translation to the Current Image Index

xmp_get_image_index(NUMBER,INDEX,CUR_INDEX,NODE_DESC)

Description. Translate image indices to the current image indices.

Class. Subroutine.

Arguments.

NUMBER shall be a scalar of type default integer. It is an INTENT(IN) argument.

INDEX shall be a rank-one array of type default integer. The size of **INDEX** shall be greater than or equal to the value of **NUMBER**. It is an INTENT(IN) argument. The value of each element of **INDEX** shall be within the range one to the size of the node array specified in **NODE_DESC**.

CUR_INDEX shall be a rank-one array of type default integer. The size of **CUR_INDEX** shall be greater than or equal to the value of **NUMBER**. It is an INTENT(OUT) argument. If the current image index corresponding to the element of the node-array specified in **NODE_DESC** whose subscript order value is **INDEX(i)** exists, **CUR_INDEX(i)** is assigned the current image index; otherwise, **CUR_INDEX(i)** is assigned zero.

NODE_DESC shall be a descriptor of a node array. It is an INTENT(IN) argument.

Example. In the following code fragment, the value of `index(1:4)` is `(/1,2,3,4/)`.

XcalableMP Fortran

```
!$xmp nodes node(1:8)=**
      integer index(4)
!$xmp task on node(5:8)
        call xmp_get_image_index&
             &(4,(/5,6,7,8/),index,xmp_desc_of(node))
!$xmp end task
```

5.6 Examples of Communication between Tasks

- In the following program fragment, two tasks communicate with each other with synchronization.

XcalableMP Fortran

```
      module nodes
!$xmp nodes node(8)=**            ! A primary node array
      integer, parameter :: n=2
!$xmp nodes subnodeA(n)=node(1:n)      ! subnodeA is for taskA.
!$xmp nodes subnodeB(8-n)=node(n+1:8)  ! subnodeB is for taskB.
      endmodule

      module intertask
      use nodes
      real,save :: dA[*],dB[*]
      endmodule

      use nodes
!$xmp tasks
!$xmp task on subnodeA  ! The taskA is invoked on subnodeA.
        call taskA
!$xmp end task
!$xmp task on subnodeB  ! The taskB is invoked on subnodeB.
        call taskB
!$xmp end task
!$xmp end tasks
      end

      subroutine taskA
      use intertask
         :
      me = this_image()    ! The value of me is i on subnodeA(i).
      if(me.eq.1)then
        call xmp_get_primary_image_index&  ! The value of iyouabs
             &(1,(/1/),iyouabs,subnodeB) ! is n+1.
!$xmp   image(node)                    ! Synchronization between
          sync images(iyouabs)         ! node(1) and node(n+1).
        call exchange(dA,dB,iyouabs)
!$xmp   image(node)                    ! Synchronization between
          sync images(iyouabs)         ! node(1) and node(n+1).
      endif
      sync all                ! Synchronization within subnodeA.
      if(me.ne.1)dA = dA[1]
      sync all                ! Synchronization within subnodeA.
       :
      end

      subroutine taskB
      use intertask
         :
      me = this_image()    ! The value of me is i on subnodeB(i).
      if(me.eq.1)then
        call xmp_get_primary_image_index&  ! The value of iyouabs
             &(1,(/1/),iyouabs,subnodeA)   ! is 1.
```

```
!$xmp   image(node)                    ! Synchronization between
          sync images(iyouabs)         ! node(n+1) and node(1).
        call exchange(dB,dA,iyouabs)
!$xmp   image(node)                    ! Synchronization between
          sync images(iyouabs)         ! node(n+1) and node(1).
      endif
      sync all                 ! Synchronization within subnodeB.
      if(me.ne.1)dB = dB[1]
      sync all                 ! Synchronization within subnodeB.

      end

      subroutine exchange(mine,yours,iput)
      use nodes
      real :: mine[*],yours[*]       ! mine and yours are always
!$xmp coarray on node :: mine,yours  ! visible on node(1:8).

      yours[iput] = mine  ! node(1) puts mine to yours[n+1] and
                          ! node(n+1) puts mine to yours[1].
      end
```

- In the following program fragment, two tasks communicate with each other without one-to-one synchronization.

XcalableMP Fortran

```
!$xmp nodes node(8)=**       ! A primary node array
        :
!$xmp tasks
!$xmp   task on(node(1:n))
          call taskA(n)      ! The taskA is invoked on node(1:n)
!$xmp   end task
!$xmp   task on(node(n+1:8))
          call taskB(8-n)    ! The taskB is invoked on node(n+1:8)
!$xmp   end task
!$xmp end tasks
      end

      subroutine taskA(n)
      real,save :: yours[*],mine[*]
!$xmp nodes subnode(n)=*        ! An executing node array

      me = this_image()
      if(me.eq. NUM_IMAGES())then
         call xmp_get_primary_image_index(1,me,meabs) ! meabs=n.
         call exchange(yours,mine,meabs,meabs+1,NUM_IMAGES())
      endif
      sync all                   ! Synchronization within node(1:n).
      if(me.ne.NUM_IMAGES())mine = mine[NUM_IMAGES()]
      sync all                   ! Synchronization within node(1:n).
      end
```

```
      subroutine taskB(m)
      real,save :: yours[*],mine[*]
!$xmp nodes subnode(m)=*        ! An executing node array

      me = this_image()
      if(me.eq.1)then
         call xmp_get_abs_image_index(1,me,meabs) ! meabs=n+1.
         call exchange(yours,mine,meabs,meabs-1,NUM_IMAGES())
      endif
      sync all                  ! Synchronization within node(n+1:8).
      if(me.ne.1)mine = mine[1]
      sync all                  ! Synchronization within node(n+1:8).
      end

      subroutine exchange(yours,mine,meabs,iyouabs,nnodes)
      USE, INTRINSIC :: ISO_FORTRAN_ENV
      real :: yours[*],mine[*]
      real, save :: s[*]                      ! only for exchage.
      TYPE(LOCK_TYPE),save :: lock[*]         ! for lock.
!$xmp nodes subnode(nnodes)=*     ! An executing node array.
!$xmp nodes node(8)=**            ! The coarrays s and lock are
!$xmp coarray on node :: s,lock   ! always visible on node(1:8).

      LOCK(lock[meabs])   ! node(n) puts yours[n] to s[n] and
      s[meabs] = yours    ! node(n+1) puts yours[n+1] to s[n+1].
      UNLOCK(lock[meabs])

      LOCK(lock[iyouabs])  ! node(n) gets s[n+1] into mine[n] and
      mine = s[iyouabs]    ! node(n+1) gets s[n] into mine[n+1].
      UNLOCK(lock[iyouabs])
      end
```

5.7 [C] Coarrays in XcalableMP C

This section describes the coarray features for XcalableMP C.

5.7.1 [C] Declaration of Coarrays

Synopsis

Coarrays are declared in XcalableMP C.

Syntax

[C] *data-type variable* : *codimensions*

where *codimensions* is:

/[*int-expr*]*.../*[*]

Description

For XcalableMP C, coarrays are declared with a colon and square bracket, where *codimensions* specify the coshape of a variable.

Note that the `coarray` directive for defining a coarray in the XcalableMP specification 1.0 (page 49) is obsolete.

Restrictions

- A coarray *variable* must have a global scope.

Examples

XcalableMP C

```
#pragma xmp nodes p[16]
float x:[*];
```

A variable x that has a global scope is declared as a coarray.

5.7.2 [C] Reference of Coarrays

Synopsis

A coarray can be directly referenced or defined by any node. The target node is specified using an extended notation in XcalableMP C.

Syntax

`[C]` *variable* `:` `[`*int-expr*`]`...

Description

A sequence of `[`*int-expr*`]`'s preceded by a colon in XcalableMP C determines the image index for a coarray to be accessed. Note that the image index in XcalableMP C is 0-origin while the image index in XcalableMP Fortran is 1-origin.

A reference of coarrays can appear in the same place as an reference of normal variables in the base languages.

Examples

In the following codes, the second image ([C] image index 1/[F] image index 2) gets all values of array `B` on the first image ([C] image index 0/[F] image index 1) to array `A` on the second image.

XcalableMP C

```
int A[100]:[*], B[100]:[*];

if(xmpc_this_image() == 1){
  A[:] = B[:]:[0];
}
```

XcalableMP Fortran

```
integer :: A(100)[*], B(100)[*]

if (this_image() == 2) then
  A(:) = B(:)[1]
end if
```

5.7.3 [C] Synchronization of Coarrays

Synopsis

XcalableMP C provides synchronization functions for coarrays.

Format

```
[C]  void  xmp_sync_all(int* status)
[C]  void  xmp_sync_memory(int* status)
[C]  void  xmp_sync_image(int image, int* status)
[C]  void  xmp_sync_images(int num, int* image_set, int* status)
[C]  void  xmp_sync_images_all(int* status)
```

Description

- `xmp_sync_all` is equivalent to the `sync all` statement in Fortran 2008.
- `xmp_sync_memory` is equivalent to the `sync memory` statement in Fortran 2008.
- A combination of `xmp_sync_image`, `xmp_sync_images`, and `xmp_sync_images_all` is equivalent to the `sync images` statement in Fortran 2008.
  - `xmp_sync_image` is to synchronize one image.
  - `xmp_sync_images` is to synchronize some images.
  - `xmp_sync_images_all` is to synchronize all images.

Arguments

- The argument *status* is defined with one of the follow symbolic constants.
  - `XMP_STAT_SUCCESS`
  - `XMP_STAT_STOPPED_IMAGE`

  If an execution of the function is successful, the *status* is defined using `XMP_STAT_SUCCESS`. The condition where the *status* is defined using `XMP_STAT_STOPPED_IMAGE` is the same as that where the *status* is defined using `STAT_STOPPED_IMAGE` in Fortran 2008. These symbolic constants are defined in "xmp.h." If any other error condition occurs during the execution of these functions, the *status* is defined with a value that is different from the value of `XMP_STAT_SUCCESS` and `XMP_STAT_STOPPED_IMAGE`.
- In `xmp_sync_image`, the variable *image* determines a target image index.
- In `xmp_sync_images`, the variable *num* is a number of target images, and the variable *image_set* is an array in which the target image set is defined.

5.8 Directives for the Local-view Programming

5.8.1 [F] `local_alias` Directive

Synopsis

In XcalableMP Fortran, the `local_alias` directive declares a local data object as an alias to the local section of a mapped array.

Syntax

```
[F]  !$xmp local_alias local-array-name => global-array-name
```

Description

The LOCAL_ALIAS directive associates a non-mapped array with an explicitly mapped array. The non-mapped array is an associating local array and the explicitly mapped array is an associated global array. The shape of the associating local array is the same as that of the node-local portion of the associated global array including the shadow area. Each element of the associating local array shares the same storage unit in array-element order with that of the node-local portion of the associated global array. An associating local array and the corresponding global array always have the same allocation status. An associating local array always has the dynamic type and type parameter values of the corresponding associated global array.

An associating local array may be a coarray. An associating local array that is a coarray is an on-node coarray of the node array onto which the corresponding associated global array is mapped. All specifications and restrictions on coarrays are also applied to an associating local array that is a coarray, with the exception that an associating local array is always declared with *assumed-shape-spec-list* of the same rank as the associated global array. In particular, a processor shall ensure that an associating local array that is a coarray has the same bounds on all the images corresponding to the node array onto which the corresponding associated global array is mapped. The mapping attributes that are allowed for an associated global array are processor dependent.

Note that the base language Fortran is extended so that a deferred-shape array that is neither an allocatable array nor an array pointer is declared if it is specified as a *local-array-name* in the `local_alias` directive.

In XcalableMP C, the `address-of` operator is applied to global data substitutes for the `local_alias` directive (see 3.4).

Restrictions

- A *global-array-name* shall be the name of an explicitly mapped array declared in the same scoping unit.
- A *local-array-name* shall be the name of a non-mapped array declared in the same scoping unit.
- A *local-array-name* shall not be a dummy argument.
- An associating local array shall have the declared type and type parameters of the corresponding associated global array.
- An associating local array shall be declared with *assumed-shape-spec-list* of the same rank as the corresponding associated global array.
- A *local-array-name* shall appear in a `coarray` directive in the same scoping unit and the *node-name* in the `coarray` directive shall be the name of the node array onto which the associated global array is mapped.
- If an associated global array is a dummy argument and corresponds to an associating local array that is a coarray, the corresponding effective argument shall be an explicitly mapped array or a subobject of an explicitly mapped array whose name appears in a LOCAL_ALIAS directive, and the corresponding associating local array shall be a coarray.
- If a dummy argument is a coarray and the corresponding ultimate argument is a coarray appearing in a LOCAL_ALIAS directive, the dummy argument shall appear in a COAR-

RAY directive with a node array corresponding to a subset of the set of images that corresponds to the node array onto which the ultimate argument is mapped.

Examples

Example 1

XcalableMP Fortran

```
!$xmp nodes n(4)
!$xmp template :: t(100)
!$xmp distribute (block) onto n :: t

      real :: a(100)
!$xmp align (i) with t(i) :: a
!$xmp shadow (1) :: a

      real :: b(:)
!$xmp local_alias b => a
```

The array `a` is distributed by block onto four nodes. The node `n(2)` has its local section of 25elements (`a(25:50)`) with shadow areas of size one on both the upper and lower bounds. The local alias `b` is an array of 27 elements (`b(1:27)`) on `n(2)`. The table below shows the correspondence of each element of `a` to that of `b` on `n(2)`.

a	b
lower shadow	1
26	2
27	3
28	4
...	...
50	26
upper shadow	27

Example 2

XcalableMP Fortran

```
!$xmp nodes n(4)
!$xmp template :: t(100)
!$xmp distribute (cyclic) onto n :: t

      real :: a(100)
!$xmp align (i) with t(i) :: a

      real :: b(0:)
!$xmp local_alias b => a
```

An array `a` is distributed cyclically onto four nodes. Node `n(2)` has its local section of 25 elements (`a(2:100:4)`). The lower bound of local alias `b` is declared to be zero. As a result, `b` is an array of size 25 whose lower bound is zero (`b(0:24)`) on `n(2)`. The table below shows the correspondence of each element of a to that of `b` on `n(2)`.

a	b
2	0
6	1
10	2
...	...
98	24

Example 3

XcalableMP Fortran

```
!$xmp nodes n(4)
!$xmp template :: t(:)
!$xmp distribute (block) onto n :: t

      real, allocatable :: a(:)
!$xmp align (i) with t(i) :: a

      real :: b(:)[*]
!$xmp local_alias b => a

      ...

!$xmp template_fix :: t(128)

      allocate (a(128))

      if (me < 4) b(4) = b(4)[me +1]
```

Because the global array `a` is an allocatable array, its local alias `b` is not defined when the subroutine starts execution. `b` is defined when `a` is allocated at the `allocate` statement. Note that `b` is declared as a coarray, and can therefore be accessed in the same manner as a normal coarray.

5.8.2 `post` Construct

Synopsis

The `post` construct, in combination with the `wait` construct, specifies the point-to-point synchronization.

Syntax

[F] `!$xmp post (` *nodes-ref*, *tag* `)`
[C] `#pragma xmp post (` *nodes-ref*, *tag* `)`

Description

This construct ensures that the execution of statements that precede it is completed before statements that follow the matching `wait` construct start are executed.

A `post` construct issued with the arguments of *nodes-ref* and *tag* on a node (called a *posting node*) dynamically matches at most one `wait` construct issued with the arguments of the posting node (unless omitted) and the same value as *tag* (unless omitted) by the node specified by *nodes-ref*.

Restriction

- *nodes-ref* must represent one node.
- *tag* must be an expression of type default integer, in XcalableMP Fortran, or type `int`, in XcalableMP C.

Example

Example 1

XcalableMP Fortran

```
      S1
!$xmp post (p(2), 1)
```

XcalableMP Fortran

```
!$xmp wait (p(1), 1)
      S2
```

It is assumed that the code of the left is executed by the node `p(1)`, while that on the right is executed by node `p(2)`.

The `post` construct on the left matches the `wait` construct on the right because their *nodes-ref*s represent each other and both *tags*'s have the same value of one. These constructs ensure that no statement in `S2` is executed by `p(2)` until the execution of all statements in `S1` is completed by `p(1)`.

Example 2

XcalableMP Fortran

```
!$xmp wait
      S3
```

It is assumed that this code is executed by node `p(2)`.

The `post` construct in the left code in Example 1 may match this `wait` construct because both *nodes-ref* and *tag* are omitted in this `wait` construct.

5.8.3 `wait` Construct

Synopsis

The `wait` construct, in combination with the `post` construct, specifies a point-to-point synchronization.

Syntax

[F] `!$xmp wait` *[*`(` *nodes-ref [*, *tag]* `)` *]*
[C] `#pragma xmp wait` *[*`(` *nodes-ref [*, *tag]* `)` *]*

Description

This construct prohibits statements that follow from being executed until the execution of all statements preceding a matching `post` construct is completed on the node specified by *node-ref*.

A `wait` construct that is issued with the arguments of *nodes-ref* and *tag* on a node (called a *waiting node*) dynamically matches a `post` construct issued with the arguments of the waiting node and the same value as *tag* by the node specified by *nodes-ref*.

If *tag* is omitted, then the `wait` construct can match a `post` construct that is issued with the arguments of the waiting node and any tag by the node specified by *nodes-ref*. If both *tag* and *nodes-ref* are omitted, then the `wait` construct can match a `post` construct that is issued with the arguments of the waiting node and any tag on any node.

Restriction

- *nodes-ref* must represent one node.
- *tag* must be an expression of type default integer, in XcalableMP Fortran, or type `int`, in XcalableMP C.

5.8.4 [C] `lock/unlock` Construct

Synopsis

The `lock/unlock` constructs are equivalent to the `lock/unlock` statements in Fortran 2008.

Syntax

```
[C]  #include <xmp.h>
[C]  xmp_lock_t lock-object [, lock-object ]...
[C]  #pragma xmp lock (lock-object) [ acquired_lock (success) ] [ stat (status) ]
[C]  #pragma xmp unlock (lock-object) [ stat (status) ]
```

Please note the following points:

- The type `xmp_lock_t` is defined in "xmp.h".
- The variable *lock-object* is a coarray.
- The variable *success* is an expression of type `int`.
- The variable *status* is an expression of type `int`.

Description

The `lock` construct, in combination with the `unlock` construct, is used to control a *lock-object*. The *lock-object* must be defined as a coarray to control it on a target node. The *lock-object* must be an expression of type `xmp_lock_t`, which is an opaque object defined in "xmp.h".

If the `acquired_lock` clause is not used in the `lock` construct and the *lock-object* is locked, the node stops at the `lock` construct until the *lock-object* is unlocked by a different node. If the `acquired_lock` clause is used in the `lock` construct and the *lock-object* is locked by a different node, the node does not stop at the `lock` construct and the variable *success* is defined with the value false; `lock` construct leaves the *lock-object* unchanged. If the `acquired_lock` clause is used in the `lock` construct and the *lock-object* is unlocked, the variable *success* is defined with the value true.

The *status* is defined with one of the follow symbolic constants when executing the `lock/unlock` construct.

- `XMP_STAT_SUCCESS`
- `XMP_STAT_LOCKED`
- `XMP_STAT_UNLOCKED`
- `XMP_STAT_LOCKED_OTHER_IMAGE`

If the execution of the `lock/unlock` construct is successful, the *status* is defined with `XMP_STAT_SUCCESS`. The condition where the *status* is defined with `XMP_STAT_LOCKED`, `XMP_STAT_UNLOCKED`, or `XMP_STAT_ LOCKED_OTHER_IMAGE` is the same as that where the *status* is defined with `STAT_LOCKED`, `STAT_UNLOCKED`, or `STAT_LOCKED_OTHER_IMAGE` in Fortran 2008. These symbolic constants are defined in "xmp.h". If any other error condition occurs during the execution of these constructs, the *status* is defined with a value that is different from the value of `XMP_STAT_SUCCESS`, `XMP_STAT_LOCKED`, `XMP_STAT_UNLOCKED`, and `XMP_STAT_LOCKED_OTHER_IMAGE`.

Example

XcalableMP C

```
#include "xmp.h"

xmp_lock_t lock_obj:[*];
int A:[*], B;
#pragma xmp nodes p[2]
...
#pragma xmp lock(lock_obj:[2])
  if(xmp_node_num() == 1){
    A:[2] = B;
  }
#pragma xmp unlock(lock_obj:[2])
```

Chapter 6

Procedure Interfaces

This chapter describes the procedure interfaces, that is, how procedures are invoked and arguments are passed, in XcalableMP.

In order to achieve high composability of XcalableMP programs, it is one of the most important requirement that XcalableMP procedures can invoke procedures written in the base language with as few restrictions as possible.

6.1 General Rule

In XcalableMP, a procedure invocation is itself a local operation, and does not cause any communication or synchronization at runtime. Thus, a node can invoke any procedure, whether written in XcalableMP or in the base language, at any point during the execution. There is no restriction on the characteristics of procedures invoked by an XcalableMP procedure, except for a few ones on its argument, which are explained below.

Local data in the actual or dummy argument list (referred to as a *local actual argument* and a *local dummy argument*, respectively) are treated by the XcalableMP compiler in the same manner as the compiler of the base language. This rule makes it possible for a local actual argument in a procedure written in XcalableMP to be associated with a dummy argument of a procedure written in the base language.

If both an actual argument and its associated dummy argument are coarrays, they must be declared on the same node set.

Implementation The XcalableMP compiler does not transform either local actual or dummy arguments, so the backend compiler of the base language can treat them in its usual way.

The rest of this chapter specifies how global data appearing in an actual and/or dummy argument list (referred to as a *global actual argument* and a *global dummy argument*, respectively) are processed by the XcalableMP compiler.

6.2 Argument Passing Mechanism in XcalableMP Fortran

Either of the following global data can be put in the actual argument list:

- an array name;
- an array element; or
- an array section that satisfies both of the following conditions:

- its subscript list is a list of zero or more colons (":") followed by zero or more *int-expr*'s;
- the subscript of the dimension having a shadow is *int-expr* unless it is the last dimension.

There are two kinds of argument association for global data in XcalableMP Fortran: one is *sequence association*, which is for global dummies that are an explicit-shape or assumed-size array, and the other is *descriptor association*, which is for all other.

6.2.1 Sequence Association of Global Data

The concept of sequence association in Fortran is extended for global actual and dummy arguments in XcalableMP as follows.

If the actual argument is an array name or an array section that satisfies the above conditions, it represents an element sequence consisting of the elements of its local section in Fortran's array element order on each node. In addition, if the actual argument is an element of a global data object, it represents an element sequence consisting of the corresponding element in the local section and each element that follows it in array element order on each node.

An global actual argument that represents an element sequence and corresponds to a global dummy argument is sequence associated with the dummy argument if the dummy argument is an explicit-shape or assumed-size array. According to this (extended) rule of sequence association, each element of the element sequence represented by the global actual argument is associated with the element of the local section of the global dummy argument that has the same position in array element order.

Sequence association is the default rule of association for global actual arguments, and it is therefore applied unless it is obvious from the interface of the invoked procedure that the corresponding dummy argument is neither an explicit-shape nor assumed-size array.

Implementation In order to implement sequence association, the name, a section, or an element of global data appearing as an actual argument is treated by the XcalableMP compiler as the base address of its local section on each node, and the global data appearing as the corresponding dummy argument is initialized at runtime so that it is composed of the local sections, each of which starts from the address received as the argument. On a node that does not have the local section corresponding to the actual argument, an unspecified value (e.g. null) is received.

Such an implementation implies that in many cases, in order to associate properly a global actual argument with the global dummy argument, their mappings (including their shadow attributes) must be identical.

Examples

Example 1 Both the actual argument `a` and the dummy argument `x` are global explicit-shape arrays, and therefore, `a` is sequence associated with `x`.

The base address of the local section of `a` is passed between these subroutines on each node. Each of the local sections of `x` starts from the received address (Figure 6.1).

XcalableMP Fortran

```
      subroutine xmp_sub1
!$xmp nodes p(4)
!$xmp template t(100)
!$xmp distribute t(block) onto p
```

```
      real a(100)
!$xmp align a(i) with t(i)
!$xmp shadow a(1:1)
      call xmp_sub2(a)
      end subroutine

      subroutine xmp_sub2(x)
!$xmp nodes p(4)
!$xmp template t(100)
!$xmp distribute t(block) onto p
      real x(100)
!$xmp align x(i) with t(i)
!$xmp shadow x(1:1)
      ...
```

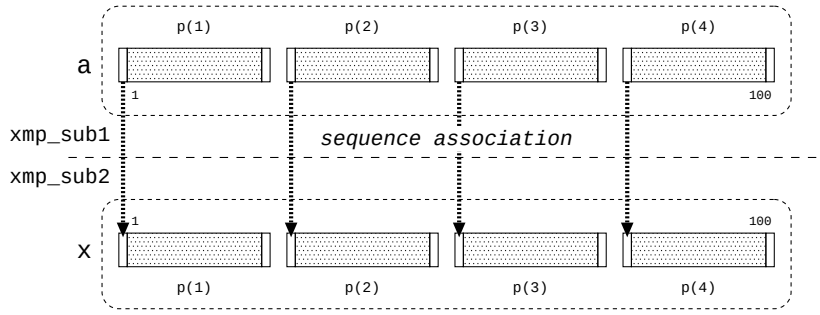

Figure 6.1: Sequence association with a global dummy argument.

Example 2 The actual argument `a` is a global explicit-shape array, and the dummy argument `x` is a local explicit-shape. Sequence association is also applied in this case.

The caller subroutine `xmp_sub1` passes the base address of the local section of `a` on each node, and the callee `f_sub2` receives it and initializes `x` with the storage starting from it (Figure 6.2).

XcalableMP Fortran

```
      subroutine xmp_sub1
!$xmp nodes p(4)
!$xmp template t(100)
!$xmp distribute t(block) onto p
      real a(100)
!$xmp align a(i) with t(i)
!$xmp shadow a(1:1)
      n = 1 + 100/4 + 1
      call f_sub2(a,n)
      end subroutine
```

Fortran

```
      subroutine f_sub2(x,n)
      real x(n)
      ...
```

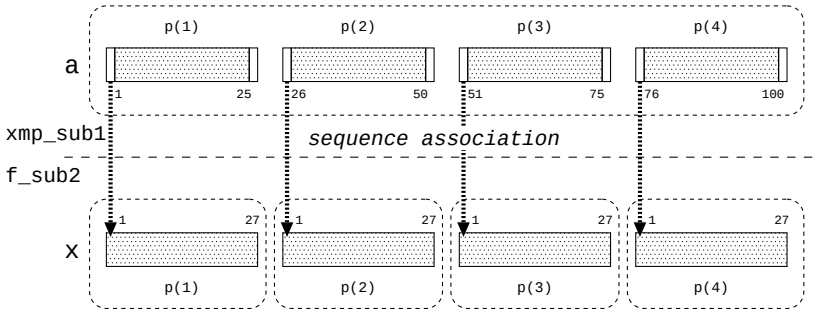

Figure 6.2: Sequence association with a local dummy argument.

Example 3 The actual argument `a(:,1)` is a contiguous section of a global data object, and the dummy argument `x` is a local explicit-shape array. Sequence association is applied in this case, but only the node `p(1)` owns the section. Hence, `f_sub2` is invoked only by `p(1)` (Figure 6.3).

XcalableMP Fortran

```
      subroutine xmp_sub1
!$xmp nodes p(4)
!$xmp template t(100,100)
!$xmp distribute t(*,block) onto p
      real a(100,100)
!$xmp align a(i,j) with t(i,j)
!$xmp shadow a(0,1:1)
      n = 100
!$xmp task on p(1)
      call f_sub2(a(:,1),n)
!$xmp end task
      end subroutine
```

Fortran

```
      subroutine f_sub2(x,n)
      real x(n)
      ...
```

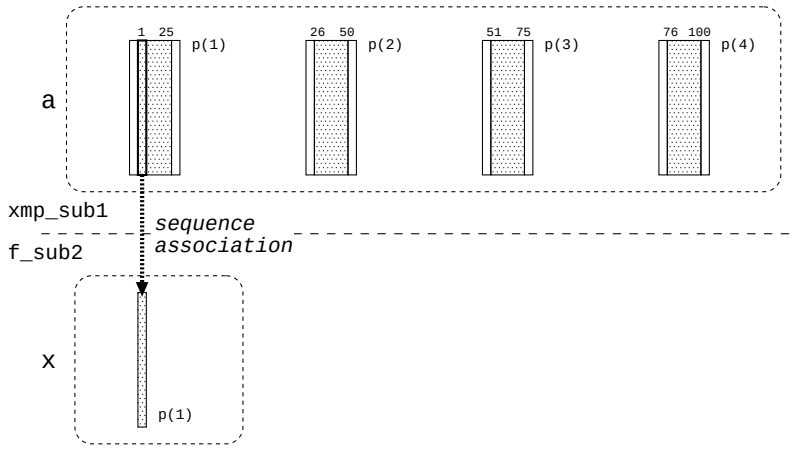

Figure 6.3: Sequence association of a section of a global data object as an actual argument with a local dummy argument.

Example 4 The actual argument `a(1)` is an element of the global data, and the dummy argument `x` is a local explicit-shape array. Sequence association is applied in this case, but only the node `p(1)` owns the element. Hence, `f_sub2` is invoked only by `p(1)`(Figure 6.4).

XcalableMP Fortran

```
      subroutine xmp_sub1
!$xmp nodes p(4)
!$xmp template t(100)
!$xmp distribute t(block) onto p
      real a(100)
!$xmp align a(i) with t(i)
!$xmp shadow a(1:1)
      n = 100/4
!$xmp task on p(1)
      call f_sub2(a(1),n)
!$xmp end task
      end subroutine
```

Fortran

```
      subroutine f_sub2(x,n)
      real x(n)
      ...
```

Example 5 Even if either the global actual or dummy argument has a full shadow, the rule of sequence association is the same in principle. Hence, the base address of the local section of `a` is passed between these subroutines on each node, and each local section of `x` starts from the received address (Figure 6.5).

6.2.2 Descriptor Association of Global Data

When the actual argument is a global data object, and it is obvious from the interface of the invoked procedure that the corresponding dummy argument is neither an explicit-shape nor assumed-size array, the actual argument is *descriptor associated* with the dummy argument. According to the descriptor association rule, the dummy argument inherits its shape and storage from the actual argument.

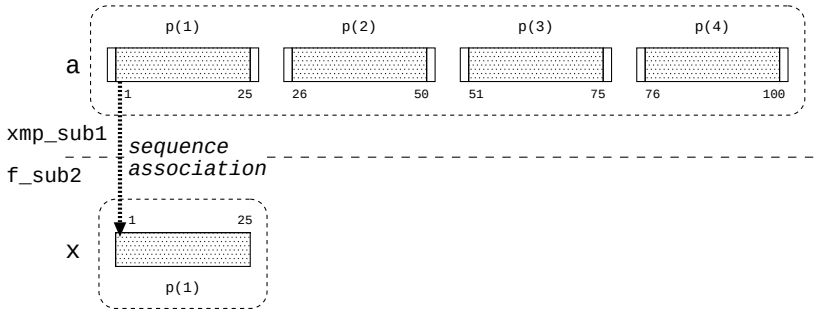

Figure 6.4: Sequence association of an element of a global data object as an actual argument with a local dummy argument.

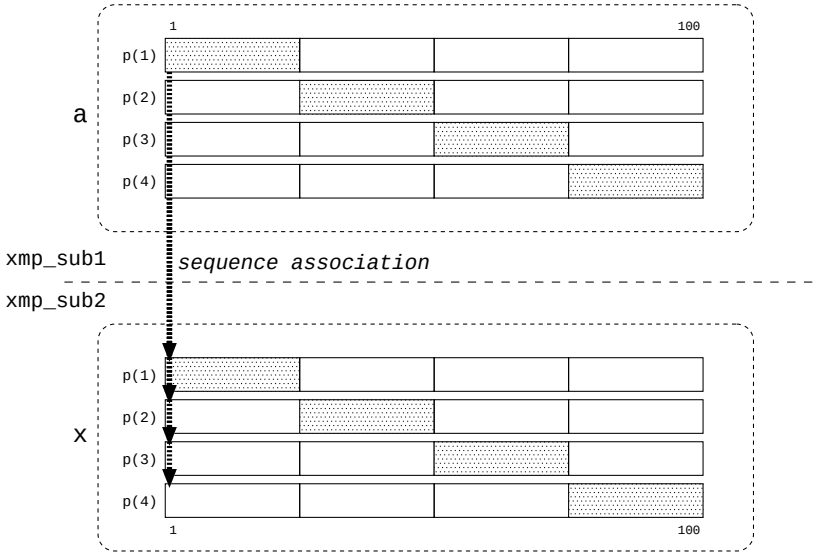

Figure 6.5: Sequence association with a global dummy argument that has a full shadow.

Implementation In order to implement the descriptor association, a global actual argument is treated by the XcalableMP compiler:

- as if it were the *global-data descriptor* of the actual array, which is an internal data structure managed by the XcalableMP runtime system to store information on a global data object (see 7.1.1) if the dummy is a global data object; or

- as it is an array representing the local section of the actual array, which is to be processed by the backend Fortran compiler in the same manner as usual data if the dummy is a local data object.

For the first case, a global dummy is initialized at runtime with a copy of the global-data descriptor received.

When an actual argument is descriptor associated with the dummy argument and their mappings are not identical, the XcalableMP runtime system may detect and report the error.

Examples

Example 1 There is an explicit interface of the subroutine `xmp_sub2` specified by an interface block in the subroutine `xmp_sub1`, from which it is found that the dummy argument `x` is a global assumed-shape array. Therefore, the global actual argument `a` is descriptor associated with the global dummy argument `x`.

It is the global-data descriptor of `a` that is passed between these subroutines. The dummy argument `x` is initialized by the XcalableMP runtime system on the basis of the information extracted from the descriptor received (Figure 6.6).

XcalableMP Fortran

```
      subroutine xmp_sub1

!$xmp nodes p(4)
!$xmp template t(100)
!$xmp distribute t(block) onto p
      real a(100)
!$xmp align a(i) with t(i)
!$xmp shadow a(1:1)

      interface
      subroutine xmp_sub2(x)
!$xmp nodes p(4)
!$xmp template t(100)
!$xmp distribute t(block) onto p
      real x(:)
!$xmp align x(i) with t(i)
!$xmp shadow a(1:1)
      end subroutine xmp_sub2
      end interface

      call xmp_sub2(a)

      end subroutine

      subroutine xmp_sub2(x)
!$xmp nodes p(4)
!$xmp template t(100)
!$xmp distribute t(block) onto p
      real x(:)
!$xmp align x(i) with t(i)
!$xmp shadow a(1:1)
      ...
```

Example 2 There is the explicit interface of the subroutine `f_sub2`, which is written in Fortran, specified by an interface block in the subroutine `xmp_sub1`, and the dummy argument `x` is a local (i.e., non-mapped) assumed-shape array. Therefore, the global actual argument `a` is descriptor associated with the local dummy argument `x`.

The global actual argument is replaced with its local section by the XcalableMP compiler, and the association of the local section with the dummy argument is to be processed by the backend Fortran compiler in the same manner as usual data (Figure 6.7).

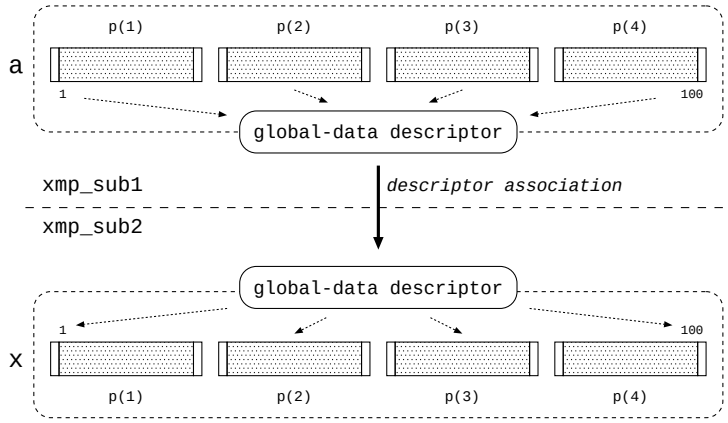

Figure 6.6: Descriptor association with a global dummy argument.

XcalableMP Fortran

```
      subroutine xmp_sub1

!$xmp nodes p(4)
!$xmp template t(100)
!$xmp distribute t(block) onto p
      real a(100)
!$xmp align a(i) with t(i)
!$xmp shadow a(1:1)

      interface
      subroutine f_sub2(x)
      real x(:)
      end subroutine f_sub2
      end interface

      call f_sub2(a)

      end subroutine
```

Fortran

```
      subroutine f_sub2(x)
      real x(:)
      ...
```

6.3 Argument-Passing Mechanism in XcalableMP C

When an actual argument is a global data object, it is passed by the address of its local section. When a dummy argument is a global data object , an address is received and used as the base address of each of its local sections.

Implementation The name of a global data object appearing as an actual argument is treated by the XcalableMP compiler as the pointer to the first element of its local section on each

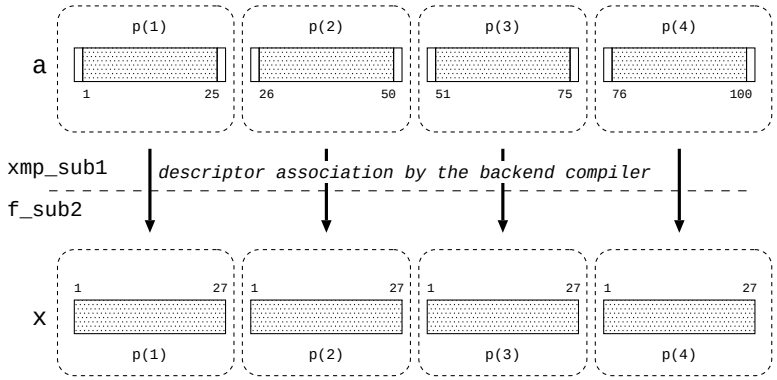

Figure 6.7: Descriptor association with a local dummy argument.

node. On a node onto which no part of the global data object is mapped, the pointer is set to an unspecified value (e.g., null). Note that an element of a global data object in the actual argument list is treated in the same manner as those in other usual statements because an array element is passed by value, as in C.

The name of a global data object appearing as a dummy argument is treated by the XcalableMP compiler as the pointer to the first element of its local section on each node. As a result, it is initialized at runtime so that it is composed of the local sections on the executing nodes.

Such an implementation implies that in many cases, in order to pass properly a global actual argument to the corresponding global dummy argument, their mappings (including their shadow attributes) must be identical.

Examples

Example 1 The global actual argument `a` is treated by the XcalableMP compiler as the pointer to the first element of its local section, which is passed to the callee, on each node.

The global dummy argument `x` is initialized so that each of its local sections starts from the address held by the received pointer (Figure 6.8).

XcalableMP C

```
void xmp_func1()
{
#pragma xmp nodes p[4]
#pragma xmp template t[100]
#pragma xmp distribute t[block] onto p
  float a[100];
#pragma xmp align a[i] with t[i]
#pragma xmp shadow a[1:1]

  xmp_func2(a);
}

void xmp_func2(float x[100])
{
#pragma xmp nodes p[4]
#pragma xmp template t[100]
#pragma xmp distribute t[block] onto p
```

```
#pragma xmp align x[i] with t[i]
#pragma xmp shadow a[1:1]
  ...
```

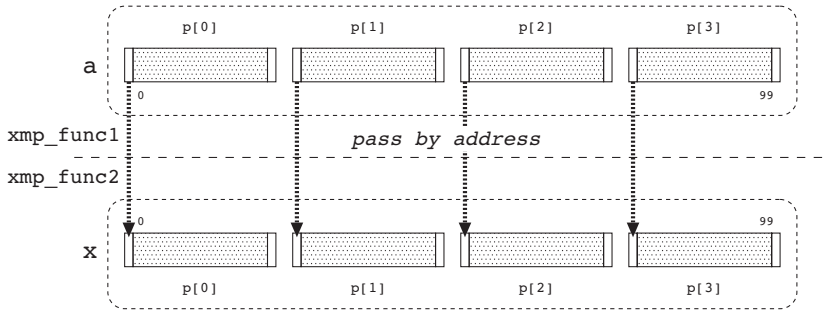

Figure 6.8: Passing to a global dummy argument.

Example 2 The global actual argument `a` is treated by the XcalableMP compiler as the pointer to the first element of its local section, which is passed to the callee, on each node.

The local dummy argument `x` on each node starts from the address held by the received pointer (Figure 6.9).

XcalableMP C

```
void xmp_func1()
{
#pragma xmp nodes p[4]
#pragma xmp template t[100]
#pragma xmp distribute t[block] onto p
  float a[100];
#pragma xmp align a[i] with t[i]
#pragma xmp shadow a[1:1]

  c_func2(a);
}
```

C

```
void c_func2(float x[27])
{
  ...
```

Example 3 The actual argument `a[0]` is an element of a global data object, and the dummy argument `x` is a scalar, where the normal argument-passing rule of C for variables of a basic type (i.e., "pass-by-value") is applied. However, only the node `p[0]` owns the element. Hence, `c_func2` is invoked only by `p[0]` (Figure 6.10).

XcalableMP C

```
void xmp_func1()
{
#pragma xmp nodes p[4]
#pragma xmp template t[100]
#pragma xmp distribute t[block] onto p
```

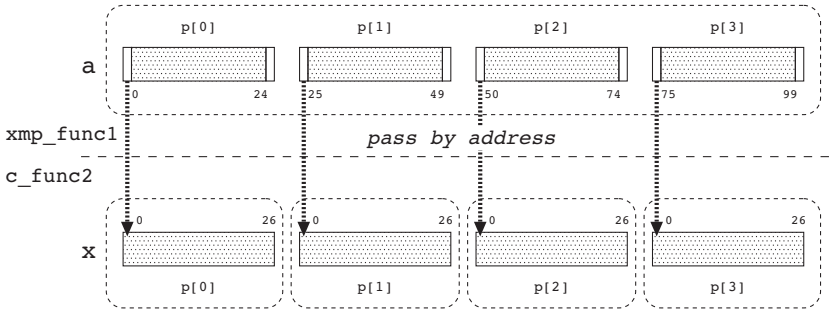

Figure 6.9: Passing to a local dummy argument.

```
  float a[100];
#pragma xmp align a[i] with t[i]
#pragma xmp shadow a[1:1]

#pragma xmp task on p[0]
  c_func2(a[0]);
}
```

C

```
void c_func2(float x)
{
  ...
```

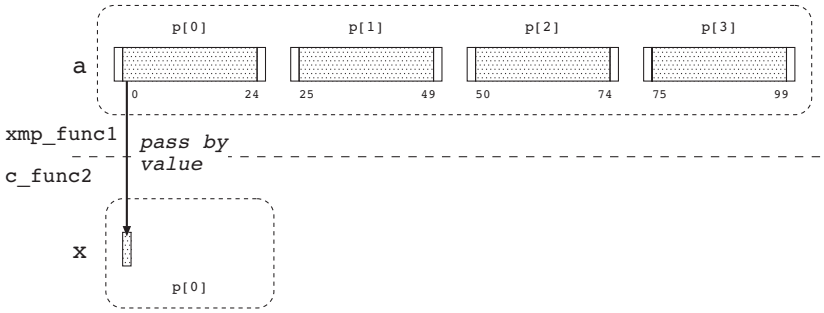

Figure 6.10: Passing an element of a global data object as an actual argument to a local dummy argument.

Chapter 7

Intrinsic and Library Procedures

This specification defines various procedures that perform a system inquiry, synchronization, computation, etc. The procedures are provided as intrinsic procedures in XcalableMP Fortran, and as library procedures in XcalableMP C.

7.1 Intrinsic Functions

7.1.1 xmp_desc_of

Format

[F] type(xmp_desc) xmp_desc_of(xmp_entity)

Note that xmp_desc_of is an intrinsic function in XcalableMP Fortran or a built-in operator in XcalableMP C. For the xmp_desc_of operator, refer to section 3.6.

Synopsis

xmp_desc_of returns a descriptor to retrieve information of the specified global array, template, or node array. The resulting descriptor can be used as an input argument of mapping inquiry functions.

The type of descriptors, type(xmp_desc), in XcalableMP Fortran, and xmp_desc_t, in XcalableMP C, is implementation-defined, and it is defined in a Fortran module named xmp_lib or a Fortran include file named xmp_lib.h.

Arguments

The argument or operand xmp_entity is the name of either a global array, a template, or a node array.

7.2 System Inquiry Functions

- xmp_all_node_num
- [C] xmpc_all_node_num
- xmp_all_num_nodes
- xmp_node_num
- [C] xmpc_node_num

- [C] xmpc_this_image
- xmp_num_nodes
- xmp_num_images
- xmp_wtime
- xmp_wtick

7.2.1 xmp_all_node_num

Format

```
[F]  integer function  xmp_all_node_num()
[C]  int               xmp_all_node_num(void)
```

Synopsis

The xmp_all_node_num routine returns the node number, within the entire node set, of the node that calls xmp_all_node_num.

Arguments

none.

7.2.2 [C] xmpc_all_node_num

Format

```
[C]  int  xmpc_all_node_num(void)
```

Synopsis

The xmpc_all_node_num routine returns the node number -1, within the entire node set, of the node that calls xmpc_all_node_num.

Arguments

none.

7.2.3 xmp_all_num_nodes

Format

```
[F]  integer function  xmp_all_num_nodes()
[C]  int               xmp_all_num_nodes(void)
```

Synopsis

The xmp_all_num_nodes routine returns the number of nodes in the entire node set.

Arguments

none.

7.2.4 xmp_node_num

Format

```
[F]  integer function  xmp_node_num()
[C]  int               xmp_node_num(void)
```

Synopsis

The xmp_node_num routine returns the node number, within the current executing node set, of the node that calls xmp_node_num.

Arguments

none.

7.2.5 [C] xmpc_node_num

Format

```
[C]  int  xmpc_node_num(void)
```

Synopsis

The xmpc_node_num routine returns the node number -1, within the current executing node set, of the node that calls xmpc_node_num.

Arguments

none.

7.2.6 [C] xmpc_this_image

Format

```
[C]  int  xmpc_this_image(void)
```

Synopsis

The xmpc_this_image routine is identical to the xmpc_node_num routine.

Arguments

none.

7.2.7 xmp_num_nodes

Format

```
[F]  integer function  xmp_num_nodes()
[C]  int               xmp_num_nodes(void)
```

Synopsis

The xmp_num_nodes routine returns the number of the executing nodes.

Arguments

none.

7.2.8 xmp_num_images

Format

```
[F]  integer function  xmp_num_images()
[C]  int               xmp_num_images(void)
```

Synopsis

The `xmp_num_images` routine is identical to the `xmp_num_nodes` routine.

Arguments

none.

7.2.9 xmp_wtime

Format

```
[F]  double precision function  xmp_wtime()
[C]  double                     xmp_wtime(void)
```

Synopsis

The `xmp_wtime` routine returns elapsed wall-clock time in seconds since some time in the past. The "time in the past" is guaranteed not to change during the life of the process. There is no requirement that different nodes return "the same time."

Arguments

none.

7.2.10 xmp_wtick

Format

```
[F]  double precision function  xmp_wtick()
[C]  double                     xmp_wtick(void)
```

Synopsis

The `xmp_wtick` routine returns the resolution of the timer used by `xmp_wtime`. It returns a double-precision value that is equal to the number of seconds between successive clock ticks.

Arguments

none.

7.3 [C] Execution Control Functions

7.3.1 xmp_exit

Format

```
[C]  void  xmp_exit(int status)
```

Synopsis

`xmp_exit` terminates an XcalableMP program normally. The value of the argument `status` returned to the host environment is the same as that by the `exit` standard library function of the base language.

`xmp_exit` must be collectively invoked by every node in the entire node set; otherwise, the behavior is undefined.

Arguments

The argument `status` is a status code to be returned to the host environment.

7.4 Synchronization Functions

7.4.1 xmp_test_async

```
[F]  logical function  xmp_test_async(async_id)
     integer           async_id

[C]  int               xmp_test_async(int async_id)
```

Synopsis

The `xmp_test_async` routine returns `.true.` in XcalableMP Fortran, or `1` in XcalableMP C, if an asynchronous communication specified by the argument `async_id` is complete; otherwise, it returns `.false.` or `0`.

Arguments

The argument `async_id` is an integer expression that specifies an asynchronous communication initiated by a global communication construct with the `async` clause.

7.5 Memory Allocation Functions

7.5.1 [C] xmp_malloc

```
void*  xmp_malloc(xmp_desc_t d, size_t size0, size_t size1, ...)
```

Synopsis

The `xmp_malloc` routine allocates storage for the local section of a global array of size `size0`×`size1`×... that is associated with the descriptor specified by `d`, and returns the pointer to it on each node. For an example of `xmp_malloc`, refer to section 3.5.

Arguments

- `d` is the descriptor associated with the pointer to a global array to be allocated.
- `size0`, `size1`, ... are the sizes of the dimensions of the global array to be allocated.

7.6 Mapping Inquiry Functions

All mapping inquiry functions are specified as integer functions. These functions return zero upon success and an implementation-defined negative integer value upon failure.

7.6.1 xmp_nodes_ndims

Format

```
[F]  integer function  xmp_nodes_ndims(d, ndims)
     type(xmp_desc)    d
     integer           ndims
[C]  int               xmp_nodes_ndims(xmp_desc_t d, int *ndims)
```

Synopsis

The `xmp_nodes_ndims` function provides the rank of the target node array.

Input Arguments

- `d` is a descriptor of a node array.

Output Arguments

- `ndims` is the rank of the node array specified by `d`.

7.6.2 xmp_nodes_index

Format

```
[F]  integer function  xmp_nodes_index(d, dim, index)
     type(xmp_desc)    d
     integer           dim
     integer           index
[C]  int               xmp_nodes_index(xmp_desc_t d, int dim, int *index)
```

Synopsis

The `xmp_nodes_index` function provides the indices of the executing node in the target node array.

Input Arguments

- `d` is a descriptor of a node array.
- `dim` is the target dimension of the node array.

Output Arguments

- `index` is an index of the target dimension of the node array specified by `d`.

7.6.3 xmp_nodes_size

Format

```
[F]  integer function  xmp_nodes_size(d, dim, size)
     type(xmp_desc)    d
     integer           dim
     integer           size
[C]  int               xmp_nodes_size(xmp_desc_t d, int dim, int *size)
```

Synopsis

The `xmp_nodes_size` function provides the size of each dimension of the target node array.

Input Arguments

- `d` is a descriptor of a node array.
- `dim` is the target dimension of the node array.

Output Arguments

- `size` is the extent of the target dimension of the node array specified by t d.

7.6.4 xmp_nodes_attr

Format

```
[F]  integer function  xmp_nodes_attr(d, attr)
     type(xmp_desc)    d
     integer           attr
[C]  int               xmp_nodes_attr(xmp_desc_t d, int *attr)
```

Synopsis

The `xmp_nodes_attr` function provides the attribute of the target node array. The output value of the argument `attr` is one of:

`XMP_ENTIRE_NODES`	(Entire nodes)
`XMP_EXECUTING_NODES`	(Executing nodes)
`XMP_EQUIVALENCE_NODES`	(Equivalence nodes)

These are named constants that are defined in module `xmp_lib` and in the include file `xmp_lib.h` in XcalableMP Fortran, and symbolic constants that are defined in the header file `xmp.h` in XcalableMP C.

Input Arguments

- `d` is a descriptor of a node array.

Output Arguments

- `attr` is an attribute of the target node array specified by `d`.

7.6.5 xmp_nodes_equiv

Format

```
[F]  integer function  xmp_nodes_equiv(d, dn, lb, ub, st)
     type(xmp_desc)    d
     type(xmp_desc)    dn
     integer           lb(*)
     integer           ub(*)
     integer           st(*)
[C]  int               xmp_nodes_equiv(xmp_desc_t d, xmp_desc_t *dn,
                                       int lb[], int ub[], int st[])
```

Synopsis

The `xmp_nodes_equiv` function provides the descriptor of a node array as well as a subscript list that represents a node set that is assigned to the target node array in the `nodes` directive. This function returns with a failure when the target node array is not declared as equivalenced.

Input Arguments

- `d` is a descriptor of a node array.

Output Arguments

- `dn` is the descriptor of the referenced node array if the target node array is declared as equivalenced; otherwise, `dn` is set to undefined.
- `lb` is a one-dimensional integer array the extent of which must be more than or equal to the rank of the referenced node array. The i-th element of `lb` is set to the lower bound of the i-th subscript of the node reference unless it is "`*`", or to undefined otherwise.
- `ub` is a one-dimensional integer array the extent of which must be more than or equal to the rank of the referenced node array. The i-th element of `ub` is set to the upper bound of the i-th subscript of the node reference unless it is "`*`", or to undefined otherwise.
- `st` is a one-dimensional integer array the extent of which must be more than or equal to the rank of the referenced node array. The i-th element of `st` is set to the stride of the i-th subscript of the node reference unless it is "`*`", or to zero otherwise.

7.6.6 xmp_template_fixed

Format

```
[F]  integer function  xmp_template_fixed(d, fixed)
     type(xmp_desc)    d
     logical           fixed
[C]  int               xmp_template_fixed(xmp_desc_t d, int *fixed)
```

Synopsis

The `xmp_template_fixed` function provides the logical value that shows whether the template is fixed or not.

Input Arguments

- `d` is a descriptor of a template.

Output Arguments

- `fixed` is set to true in XcalableMP Fortran and an implementation-defined non-zero integer value in XcalableMP C if the template specified by `d` is fixed; otherwise, it is set to false in XcalableMP Fortran and zero in XcalableMP C.

7.6.7 xmp_template_ndims

Format

```
[F]  integer function  xmp_template_ndims(d, ndims)
     type(xmp_desc)    d
     integer           ndims
[C]  int               xmp_template_ndims(xmp_desc_t d, int *ndims)
```

Synopsis

The `xmp_template_ndims` function provides the rank of the target template.

Input Arguments

- `d` is a descriptor of a template.

Output Arguments

- `ndims` is the rank of the template specified by `d`.

7.6.8 xmp_template_lbound

Format

```
[F]  integer function  xmp_template_lbound(d, dim, lbound)
     type(xmp_desc)    d
     integer           dim
     integer           lbound
[C]  int               xmp_template_lbound(xmp_desc_t d, int dim, int *lbound)
```

Synopsis

The `xmp_template_lbound` function provides the lower bound of each dimension of the template. This function returns with a failure when the lower bound is not fixed.

Input Arguments

- `d` is a descriptor of a template.
- `dim` is the target dimension of the template.

Output Arguments

- `lbound` is the lower bound of the target dimension of the template specified by `d`. When the lower bound is not fixed, it is set to undefined.

7.6.9 xmp_template_ubound

Format

```
[F]  integer function  xmp_template_ubound(d, dim, ubound)
     type(xmp_desc)    d
     integer           dim
     integer           ubound
[C]  int               xmp_template_ubound(xmp_desc_t d, int dim, int *ubound)
```

Synopsis

The `xmp_template_ubound` function provides the upper bound of each dimension of the template. This function returns with a failure when the upper bound is not fixed.

Input Arguments

- `d` is a descriptor of a template.
- `dim` is the target dimension of the template.

Output Arguments

- `ubound` is an upper bound of the target dimension of the template specified by `d`. When the upper bound is not fixed, it is set to undefined.

7.6.10 xmp_dist_format

Format

```
[F]  integer function  xmp_dist_format(d, dim, format)
     type(xmp_desc)    d
     integer           dim
     integer           format
[C]  int               xmp_dist_format(xmp_desc_t d, int dim, int *format)
```

Synopsis

The `xmp_dist_format` function provides the distribution format of a dimension of a template. The output value of the argument `format` is one of:

`XMP_NOT_DISTRIBUTED`	(not distributed)
`XMP_BLOCK`	(block distribution)
`XMP_CYCLIC`	(cyclic distribution)
`XMP_GBLOCK`	(gblock distribution)

These symbolic constants are defined in "xmp.h".

Input Arguments

- `d` is a descriptor of a template.
- `dim` is the target dimension of the template.

Output Arguments

- `format` is a distribution format of the target dimension of the template specified by `d`.

7.6.11 xmp_dist_blocksize

Format

```
[F]  integer function  xmp_dist_blocksize(d, dim, blocksize)
     type(xmp_desc)    d
     integer           dim
     integer           blocksize
[C]  int               xmp_dist_blocksize(xmp_desc_t d, int dim, int *blocksize)
```

Synopsis

The `xmp_dist_blocksize` function provides the block width of a dimension of a template.

Input Arguments

- `d` is a descriptor of a template.
- `dim` is the target dimension of the template.

Output Arguments

- `blocksize` is the block width of the target dimension of the template specified by `d`.

7.6.12 xmp_dist_gblockmap

Format

```
[F]  integer function  xmp_dist_gblockmap(d, dim, map)
     type(xmp_desc)    d
     integer           dim
     integer           map(N)
[C]  int               xmp_dist_gblockmap(xmp_desc_t d, int dim, int map[])
```

Synopsis

The `xmp_dist_gblockmap` function provides the mapping array of the `gblock` distribution.

When the `dim`-th dimension of the global array is distributed by `gblock` and its mapping array is fixed, this function returns zero; otherwise, it returns an implementation-defined negative integer value.

Input Arguments

- `d` is a descriptor of a template.
- `dim` is the target dimension of the template.

Output Arguments

- `map` is a one-dimensional integer array the extent of which is more than the size of the corresponding dimension of the node array onto which the template is distributed.

  The i-th element of `map` is set to the value of the i-th element of the target mapping array.

7.6.13 xmp_dist_nodes

Format

```
[F]  integer function  xmp_dist_nodes(d, dn)
     type(xmp_desc)    d
     type(xmp_desc)    dn
[C]  int               xmp_dist_nodes(xmp_desc_t d, xmp_desc_t *dn)
```

Synopsis

The `xmp_dist_nodes` function provides the descriptor of the node array onto which a template is distributed.

Input Arguments

- `d` is a descriptor of a template.

Output Arguments

- `dn` is the descriptor of the node array.

7.6.14 xmp_dist_axis

Format

```
[F]  integer function  xmp_dist_axis(d, dim, axis)
     type(xmp_desc)    d
     integer           dim
     integer           axis
[C]  int               xmp_dist_axis(xmp_desc_t d, int dim, int *axis)
```

Synopsis

The `xmp_dist_axis` function provides the dimension of the node array onto which a dimension of a template is distributed. This function returns with a failure when the dimension of the template is not distributed.

Input Arguments

- `d` is a descriptor of a template.
- `dim` is the target dimension of the template.

Output Arguments

- `axis` is a dimension of the node array onto which the target dimension of the template specified by `d` is distributed. When the dimension of the template is not distributed, it is set to undefined.

7.6.15 xmp_align_axis

Format

```
[F]  integer function  xmp_align_axis(d, dim, axis)
     type(xmp_desc)    d
     integer           dim
     integer           axis
[C]  int               xmp_align_axis(xmp_desc_t d, int dim, int *axis)
```

Synopsis

The `xmp_align_axis` function provides the dimension of the template with which a dimension of a global array is aligned. This function returns with a failure when the dimension of the global array is not aligned.

Input Arguments

- `d` is a descriptor of a global array.
- `dim` is the target dimension of the global array.

Output Arguments

- `axis` is the dimension of the template with which the target dimension of the global array specified by `d` is aligned. When the dimension of the global array is not aligned, or is collapsed, it is set to undefined.

7.6.16 xmp_align_offset

Format

```
[F]  integer function  xmp_align_offset(d, dim, offset)
     type(xmp_desc)    d
     integer           dim
     integer           offset
[C]  int               xmp_align_offset(xmp_desc_t d, int dim, int *offset)
```

Synopsis

The `xmp_align_offset` function provides the align offset for a dimension of a global array. This function returns with a failure when there is no offset.

Input Arguments

- `d` is a descriptor of a global array.
- `dim` is the target dimension of the global array.

Output Arguments

- `offset` is the align offset for the target dimension of the global array specified by `d`. When there is no offset, it is set to undefined.

7.6.17 xmp_align_replicated

Format

```
[F]  integer function  xmp_align_replicated(d, dim, replicated)
     type(xmp_desc)    d
     integer           dim
     logical           replicated
[C]  int               xmp_align_replicated(xmp_desc_t d, int dim, int *replicated)
```

Synopsis

The `xmp_align_replicated` function provides the logical value that shows whether or not the dimension of the template with which a global array is aligned is replicated.

Input Arguments

- `d` is a descriptor of a global array.
- `dim` is the target dimension of the template with which the global array is aligned.

Output Arguments

- `replicated` is a logical scalar, which is set to true if the dimension of the template is replicated.

7.6.18 xmp_align_template

Format

```
[F]  integer function  xmp_align_template(d, dt)
     type(xmp_desc)    d
     type(xmp_desc)    dt
[C]  int               xmp_align_template(xmp_desc_t d, xmp_desc_t *dn)
```

Synopsis

The `xmp_align_template` function provides the descriptor of the template with which a global array is aligned.

Input Arguments

- `d` is a descriptor of a global array.

Output Arguments

- `dt` is the descriptor of the template.

7.6.19 xmp_array_ndims

Format

```
[F]  integer function  xmp_array_ndims(d, ndims)
     type(xmp_desc)    d
     integer           ndims
[C]  int               xmp_array_ndims(xmp_desc_t d, int *ndims)
```

Synopsis

The `xmp_array_ndims` function provides the rank of a global array.

Input Arguments

- `d` is a descriptor of a global array.

Output Arguments

- `ndims` is the rank of the global array specified by `d`.

7.6.20 xmp_array_lshadow

Format

```
[F]  integer function  xmp_array_lshadow(d, dim, lshadow)
     type(xmp_desc)    d
     integer           dim
     integer           lshadow
[C]  int               xmp_array_lshadow(xmp_desc_t d, int dim, int *lshadow)
```

Synopsis

The `xmp_array_lshadow` function provides the size of the lower shadow of a dimension of a global array.

Input Arguments

- `d` is a descriptor of a global array.
- `dim` is the target dimension of the global array.

Output Arguments

- `lshadow` is the size of the lower shadow of the target dimension of the global array specified by `d`.

7.6.21 xmp_array_ushadow

Format

```
[F]  integer function  xmp_array_ushadow(d, dim, ushadow)
     type(xmp_desc)    d
     integer           dim
     integer           ushadow
[C]  int               xmp_array_ushadow(xmp_desc_t d, int dim, int *ushadow)
```

Synopsis

The `xmp_array_ushadow` function provides the size of the upper shadow of a dimension of a global array.

Input Arguments

- `d` is a descriptor of a global array.
- `dim` is the target dimension of the global array.

Output Arguments

- `ushadow` is the size of the upper shadow of the target dimension of the global array specified by `d`.

7.6.22 xmp_array_lbound

Format

```
[F]  integer function  xmp_array_lbound(d, dim, lbound)
     type(xmp_desc)    d
     integer           dim
     integer           lbound
[C]  int               xmp_array_lbound(xmp_desc_t d, int dim, int *lbound)
```

Synopsis

The `xmp_array_lbound` function provides the lower bound of a dimension of a global array. This function returns with a failure when the lower bound is not fixed.

Input Arguments

- `d` is a descriptor of a global array.
- `dim` is the target dimension of the global array.

Output Arguments

- `lbound` is the lower bound of the target dimension of the global array specified by `d`. When the lower bound is not fixed, it is set to undefined.

7.6.23 xmp_array_ubound

Format

```
[F]  integer function  xmp_array_ubound(d, dim, ubound)
     type(xmp_desc)    d
     integer           dim
     integer           ubound
[C]  int               xmp_array_ubound(xmp_desc_t d, int dim, int *ubound)
```

Synopsis

The `xmp_array_ubound` function provides the upper bound of a dimension of a global array. This function returns with a failure when the upper bound is not fixed.

Input Arguments

- `d` is a descriptor of a global array.
- `dim` is the target dimension of the global array.

Output Arguments

- `ubound` is the upper bound of the target dimension of the global array specified by `d`. When the upper bound is not fixed, it is set to undefined.

7.7 [F] Array Intrinsic Functions of the Base Language

The array intrinsic functions of the base language Fortran are classified into three classes: *inquiry*, *elemental*, and *transformational*.

This section specifies how these functions work in the XMP/F programs when a global array appears as an argument.

- Inquiry functions

  The inquiry functions with a global array or its subobject being an argument are regarded as inquiries about the global array, and return its "global" properties as if it were not distributed.

- Elemental functions

  The result of the elemental functions with a global array or its subobject being an argument has the same shape and mapping as the argument. Note that such a reference of these elemental functions is in effect limited to be in the `array` construct.

- Transformational functions

  It is unspecified how the transformational functions work when a global array or its subobject appears as an argument. A processor shall detect such a reference of these functions and issue a warning message for it. Some intrinsic transformational subroutines are defined in section 7.9 as alternatives to these transformational functions.

7.8 [C] Built-in Elemental Functions

Some built-in elemental functions that can operate each element of array arguments are defined in XcalableMP C. Such a built-in function accepts one or more array sections as its arguments and returns an array-valued result having the same shape and mapping as the argument. The values of the elements of the result are the same as what would have been obtained if the scalar function of the C standard library had been applied separately to the corresponding elements of each array argument.

These functions may appear on the right-hand side of an array assignment statement, and it should be preceded by the `array` directive if the array section is distributed.

Table 7.1 shows the list of built-in elemental functions in XcalableMP C. Their elementwise behavior is the same as those of the corresponding functions in the C standard library.

7.9 Intrinsic/Built-in Transformational Procedures

Some intrinsic/built-in transformational procedures are defined for the non-elemental operations of arrays.

Note that each "array argument" of the following procedures must be an array name or an array section, in XcalableMP Fortran, or an array section, in XcalableMP C, that represents the whole array.

Table 7.1: Built-in elemental functions in XcalableMP C. (The first line refers to the element type of their argument(s) and return value.)

double	float	long double
acos	acosf	acosl
asin	asinf	asinl
atan	atanf	atanl
atan2	atan2f	atan2l
cos	cosf	cosl
sin	sinf	sinl
tan	tanf	tanl
cosh	coshf	coshl
sinh	sinhf	sinhl
tanh	tanhf	tanhl
exp	expf	expl
frexp	frexpf	frexpl
ldexp	ldexpf	ldexpl
log	logf	logl
log10	log10f	log10l
fabs	fabsf	fabsl
pow	powf	powl
sqrt	sqrtf	sqrtl
ceil	ceilf	ceill
floor	floorf	floorl
fmod	fmodf	fmodl

7.9.1 `xmp_scatter`

Format

```
[F]         xmp_scatter(x, a, idx1, ..., idxn)
[C]  void   xmp_scatter(x[:]..., a[:]..., idx1[:]..., ..., idxn[:]...)
```

Synopsis

The `xmp_scatter` procedure copies the value of each element of an array `a` to the corresponding element of an array `x` that is determined by vectors `idx1`, ..., `idxn`.

This procedure produces the same result as the following Fortran assignment statement when `x`, `a`, and `idx1`, ..., `idxn` are not mapped.

```
x(idx1(:,:,...), ..., idxn(:,:,...)) = a(:,:,...)
```

If any of the vectors `idx1`, ..., `idxn` have two or more elements with the same value, the behavior and the result of `xmp_scatter` is unspecified.

Output Arguments

- `x` is an array of any type, shape, and mapping.

Input Arguments

- `a` is an array of the same type as `x` and any shape and mapping.
- `idx1`, ..., `idxn` are integer arrays of the same shape and mapping as `a`. The number of `idx`'s is equal to the rank of `x`.

7.9.2 xmp_gather

Format

```
[F]         xmp_gather(x, a, idx1, ..., idxn)
[C]  void   xmp_gather(x[:]..., a[:]..., idx1[:]..., ..., idxn[:]...)
```

Synopsis

The `xmp_gather` procedure copies the value of each element of an array `a` determined by vectors `idx1`, ..., `idxn` to the corresponding element of an array `x`.

This procedure produces the same result as the following Fortran assignment statement when `x`, `a`, and `idx1`, ..., `idxn` are not mapped.

```
x(:,:,...) = a(idx1(:,:,...), ..., idxn(:,:,...))
```

Output Arguments

- `x` is an array of any type, shape, and mapping.

Input Arguments

- `a` is an array of the same type as `x` and any shape and mapping.
- `idx1`, ..., `idxn` are integer arrays of the same shape and mapping as `x`. The number of `idx`'s is equal to the rank of `a`.

7.9.3 xmp_pack

Format

```
[F]         xmp_pack(v, a, [mask])
[C]  void   xmp_pack(v[:], a[:]..., [mask[:]...])
```

Synopsis

The `xmp_pack` procedure packs all of the elements of an array `a`, if `mask` is not specified, or the elements selected by `mask`, to a vector `v` according to the array element order of the base language.

Output Arguments

- `v` is a one-dimensional array of any type, size, and mapping.

Input Arguments

- `a` is an array of the same type as `v` and any shape and mapping.
- (optional) `mask` is an array of default logical, in XcalableMP Fortran, or of type _Bool, in XcalableMP C, that has the same shape and mapping as `a`.

7.9.4 xmp_unpack

Format

```
[F]         xmp_unpack(a, v, [mask])
[C]  void   xmp_unpack(a[:]..., v[:], [mask[:]...])
```

Synopsis

The xmp_unpack procedure unpacks a vector v to all the elements of an array a, if mask is not specified, or the elements selected by a mask mask according to the array element order of the base language.

Output Arguments

- a is an array of any type, shape, and mapping.

Input Arguments

- v is a one-dimensional array of the same type of a and any shape and mapping.
- (optional) mask is an array of default logical, in XcalableMP Fortran, or of type _Bool, in XcalableMP C, that has the same shape and mapping as a.

7.9.5 xmp_transpose

Format

```
[F]         xmp_transpose(x, a, opt)
[C]  void   xmp_transpose(x[:][:], a[:][:], int opt)
```

Synopsis

The xmp_transpose procedure sets the result obtained by transposing a matrix a to a matrix x.

Output Arguments

- x is a two-dimensional array of any type, shape, and mapping.

Input Arguments

- a is a two-dimensional array of the same type as x and any mapping. The extent of the first dimension is equal to that of the second dimension of x, and the extent of the second dimension is equal to that of the first dimension of x.
- opt is an integer scalar. If opt is 0, the value of a remains unchanged after calling this procedure. If opt is 1, the value may be changed.

7.9.6 xmp_matmul

Format

```
[F]         xmp_matmul(x, a, b)
[C]  void   xmp_matmul(x[:][:], a[:][:], b[:][:])
```

Synopsis

The `xmp_matmul` procedure computes the product of matrices `a` and `b`, and it sets the result to a matrix `x`.

Output Arguments

- `x` is a two-dimensional array of any numerical type, shape and mapping.

Input Arguments

- `a` is a two-dimensional array of the same type of `x` and any mapping. The extent of the first dimension is equal to that of `x`.
- `b` is a two-dimensional array of the same type of `x` and any mapping. The extent of the first dimension is equal to that of the second dimension of `a`, and the extent of the second dimension is equal to that of `x`.

7.9.7 xmp_sort_up

Format

```
[F]        xmp_sort_up(v1, v2)
[C]  void  xmp_sort_up(v1[:], v2[:])
```

Synopsis

The `xmp_sort_up` procedure sets the result obtained by sorting elements of a vector `v2` in ascending order to a vector `v1`.

Output Arguments

- `v1` is a one-dimensional array of any numerical type, shape, and mapping.

Input Arguments

- `v2` is a one-dimensional array of the same type, shape, and mapping as `v1`.

7.9.8 xmp_sort_down

Format

```
[F]        xmp_sort_down(v1, v2)
[C]  void  xmp_sort_down(v1[:], v2[:])
```

Synopsis

The `xmp_sort_down` procedure sets the result obtained by sorting elements of a vector `v2` in descending order to a vector `v1`.

Output Arguments

- `v1` is a one-dimensional array of any numerical type, shape and mapping.

Input Arguments

- `v2` is a one-dimensional array of the same type, shape, and mapping as `v1`.

Chapter 8

OpenMP in XcalableMP Programs

The usage of OpenMP directives in XcalableMP programs is subjected to the following basic rule.

- XcalableMP directives and the invocation of an XcalableMP intrinsic/built-in procedure should be single-threaded, and they may therefore be placed in the sequential part, or one of the `single`, `master`, or `critical` regions that are closely nested inside a `parallel` region whose parent thread is the initial thread;
- with the exception that the XcalableMP's `loop` directive that controls a loop can be placed immediately inside the OpenMP's parallel loop directive (`parallel do` for Fortran and `parallel for` for C), which controls the identical loop.

The behavior of coarray references in a `parallel` region is implemetation-defined.

Examples

Assume that the following codes are placed in the sequential part of the program.

XcalableMP C

```
#pragma omp parallel for
for (...){
  #pragma xmp barrier  // NG because not single-threaded
}
```

XcalableMP C

```
#pragma omp parallel for
for (...){
  #pragma omp single
  {
    #pragma xmp barrier  // OK because single-threaded
                         // (inside a single region)
  }
}
```

XcalableMP C

```
#pragma omp parallel for
#pragma xmp loop  // OK because immediately nested
for (...){
  ...
}
```

XcalableMP C

```
#pragma xmp loop  // OK because single-threaded (not nested)
#pragma omp parallel for
for (...){
  ...
}
```

XcalableMP C

```
#pragma xmp loop  // OK because single threaded (not nested)
for (...){
  #pragma omp parallel for
  for (...) { ... }
}
```

XcalableMP C

```
#pragma omp parallel for
for (...){
  #pragma xmp loop  // NG because not immediately nested
  for (...) { ... }
}
```

Bibliography

[1] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.1", http://www.openmp.org/mp-documents/OpenMP3.1.pdf (2011).

[2] High Performance Fortran Forum, "High Performance Fortran Language Specification Version 2.0", http://hpff.rice.edu/versions/hpf2/hpf-v20.pdf (1997).

[3] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 2.2", http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf (2009).

[4] Japan Association of High Performance Fortran, "HPF/JA Language Specification", http://www.hpfpc.org/jahpf/spec/hpfja-v10-eng.pdf (1999).

[5] Yuanyuan Zhang, Hidetoshi Iwashita, Kuninori Ishii, Masanori Kaneko, Tomotake Nakamura, and Kohichiro Hotta, "Hybrid Parallel Programming on SMP Clusters Using XP-Fortran and OpenMP", Proceedings of the International Workshop on OpenMP (IWOMP 2010), Vol. 6132 of Lecture Notes in Computer Science, pp. 133–148, Springer (2010).

[6] Hidetoshi Iwashita, Naoki Sueyasu, Sachio Kamiya, and Matthijs van Waveren, "VPP Fortran and the design of HPF/JA extensions", Concurrency and Computation — Practice & Experience, Vol. 14, No. 8–9, pp. 575–588, Wiley (2002).

[7] Jinpil Lee, Mitsuhisa Sato, and Taisuke Boku, "OpenMPD: A Directive-Based Data Parallel Language Extension for Distributed Memory Systems", Proceedings of the 2008 International Conference on Parallel Processing, pp. 121-128 (2008).

Appendix A

Programming Interface for MPI

This chapter describes the programming interface for MPI, which is widely used for parallel programming in cluster computing. Users can introduce MPI functions to XcalableMP using the interface.

XcalableMP provides the following user API functions to mix MPI functions with XcalableMP.

- `xmp_get_mpi_comm`
- `xmp_init_mpi`
- `xmp_finalize_mpi`

A.1 xmp_get_mpi_comm

Format

```
[F]  integer function  xmp_get_mpi_comm()
[C]  MPI_Comm          xmp_get_mpi_comm(void)
```

Synopsis

`xmp_get_mpi_comm` returns the handle of the communicator associated with the executing node set.

Arguments

none.

A.2 xmp_init_mpi

Format

```
[F]        xmp_init_mpi()
[C]  void  xmp_init_mpi(int *argc, char ***argv)
```

Synopsis

`xmp_init_mpi` initializes the MPI execution environment.

Arguments

In XcalableMP C, the command-line arguments `argc` and `argv` should be given to `xmp_init_mpi`.

A.3 xmp_finalize_mpi

Format

```
[F]        xmp_finalize_mpi()
[C]  void  xmp_finalize_mpi(void)
```

Synopsis

`xmp_finalize_mpi` terminates the MPI execution environment.

Arguments

none.

Example

XcalableMP C

```
#include <stdio.h>
#include "mpi.h"
#include "xmp.h"

#pragma xmp nodes p[4]

int main(int argc, char *argv[]) {
  xmp_init_mpi(&argc, &argv)

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

#pragma xmp task on p[1:2]
{
  MPI_Comm comm = xmp_get_mpi_comm(); // get the MPI communicator of p[1:2]

  int rank, size;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);
}

  xmp_finalize_mpi();

  return 0;
}
```

Appendix B

Interface to Numerical Libraries

This chapter describes the XcalableMP interfaces to existing MPI parallel libraries, which is effective to achieve high productivity and performance of XcalableMP programs.

B.1 Interface Design

The recommended design of the interface is as follows:

- Numerical library routines can be invoked by an XcalableMP procedure through an interface procedure (Figure B.1).

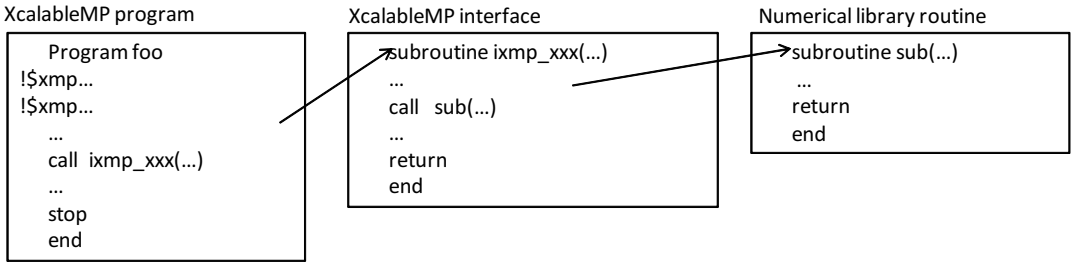

Figure B.1: Invocation of a library routine using an interface procedure.

- When the numerical library routine requires information regarding a global array, the interface extracts it from the descriptor using query routines provided by XcalableMP, and passes it to the numerical library routine as an argument.

- The interface does not affect the behavior of numerical library routines except for restrictions concerning the XcalableMP specification.

B.2 Extended Mapping Inquiry Functions

In this section, the extended mapping inquiry functions, which are implementation-defined, are shown. Specifications of the functions below are obtained from the Omni XcalableMP compiler (`http://www.xcalablemp.org/download.html`).

B.2.1 xmp_array_gtol

```
[F]  integer function  xmp_array_gtol(d, dim, g_idx, l_idx)
     type(xmp_desc)    d
     integer           dim
     integer           g_idx
     integer           l_idx

[C]  void              xmp_array_gtol(xmp_desc_t d, int dim, int g_idx, int* l_idx)
```

Synopsis

The `xmp_array_gtol` function translates a global index specified by `g_idx` of a global array specified by `d` into the corresponding index of its local section, and sets it to an array specified by `l_idx`. If the element of the specified index does not reside in the caller of the function, the resulting array is set to an unspecified value.

Input Arguments

- `d` is a descriptor of the global array.
- `dim` is the target dimension of the global array.
- `g_idx` is an index of the global array.

Output Argument

- `l_idx` is an index of the local array.

B.2.2 xmp_array_lsize

Format

```
[F]  integer function  xmp_array_lsize(d, dim, lsize)
     type(xmp_desc)    d
     integer           dim
     integer           lsize
[C]  int               xmp_array_lsize(xmp_desc_t d, int dim, int *lsize)
```

Synopsis

The `xmp_array_lsize` function provides the local size of each dimension of the target global array. Note that the local size does not include the size of the shadow.

Input Arguments

- `d` is a descriptor of a global array.
- `dim` is the target dimension of the global array.

Output Argument

- `lsize` is the local size of the target dimension of the global array.

B.2.3 xmp_array_laddr

Format

```
[C]  int  xmp_array_laddr(xmp_desc_t d, void **laddr)
```

Synopsis

The `xmp_array_laddr` function provides the local address of the target global array.

Input Arguments

- `d` is a descriptor of a global array.

Output Arguments

- `laddr` is the local address of the target global array.

B.2.4 xmp_array_lda

Format

```
[F]  integer function  xmp_array_lda(d, lda)
     type(xmp_desc)    d
     integer           lda
[C]  int               xmp_array_lda(xmp_desc_t d, int* lda)
```

Synopsis

The `xmp_array_lda` function provides the leading dimension of the two-dimensional global array. This function is used to call numerical libraries, such as BLAS.

Input Argument

- `d` is a descriptor of a global array, which must be a two-dimensional array.

Output Argument

- `lda` is a leading dimension of the target global array.

B.3 Example

This section shows the interface to ScaLAPACK as an example of the XcalableMP interface to numerical libraries.

ScaLAPACK is a linear algebra library for distributed-memory. Communication processes in the ScaLAPACK routines depend on BLACS (Basic Linear Algebraic Communication Subprograms). ScaLAPACK library routines invoked from XcalableMP procedures also depend on BLACS.

Example 1 This example shows an implementation of the interface for the ScaLAPACK driver routine `pdgesv`.

XcalableMP Fortran

```
      subroutine ixmp_pdgesv(n,nrhs,a,ia,ja,da,ipiv,b,ib,jb,db,ictxt,info)

      use xmp_lib

      integer n,nrhs,ia,ja,ib,jb,ictxt,info,desca(9),descb(9),ierr
      double precision a,b
      type(xmp_desc) da,db,dta,dtb
      integer lbound_a1,ubound_a1,lbound_a2,ubound_a2
      integer blocksize_a1,blocksize_a2,lead_dim_a
      integer lbound_b1,ubound_b1,lbound_b2,ubound_b2
      integer blocksize_b1,blocksize_b2,lead_dim_b

      ierr=xmp_array_lbound(da,1,lbound_a1)
      ierr=xmp_array_ubound(da,1,ubound_a1)
      ierr=xmp_array_lbound(da,2,lbound_a2)
      ierr=xmp_array_ubound(da,2,ubound_a2)
      ierr=xmp_align_template(da,dta)
      ierr=xmp_dist_blocksize(dta,1,blocksize_a1)
      ierr=xmp_dist_blocksize(dta,2,blocksize_a2)
      ierr=xmp_array_lead_dim(da,1,lead_dim_a)

      ierr=xmp_array_lbound(db,1,lbound_b1)
      ierr=xmp_array_ubound(db,1,ubound_b1)
      ierr=xmp_array_lbound(db,2,lbound_b2)
      ierr=xmp_array_ubound(db,2,ubound_b2)
      ierr=xmp_align_template(db,dtb)
      ierr=xmp_dist_blocksize(dtb,1,blocksize_b1)
      ierr=xmp_dist_blocksize(dtb,2,blocksize_b2)
      ierr=xmp_array_lead_dim(db,1,lead_dim_b)

      desca(1)=1
      desca(2)=ictxt
      desca(3)=ubound_a1-lbound_a1+1
      desca(4)=ubound_a2-lbound_a2+1
      desca(5)=blocksize_a1
      desca(6)=blocksize_a2
      desca(7)=0
      desca(8)=0
      desca(9)=lead_dim_a

      descb(1)=1
      descb(2)=ictxt
      descb(3)=ubound_b1-lbound_b1+1
      descb(4)=ubound_b2-lbound_b2+1
      descb(5)=blocksize_b1
      descb(6)=blocksize_b2
      descb(7)=0
      descb(8)=0
      descb(9)=lead_dim_b
```

```
      call pdgesv(n,nhrs,a,ia,ja,desca,ipiv,b,ib,jb,descb,info)

      return
      end
```

Example 2 This example shows an XcalableMP procedure using the interface of Example 1.

XcalableMP Fortran

```
      program xmptdgesv

      use xmp_lib

      double precision a(1000,1000)
      double precision b(1000)
      integer ipiv(2*1000,2)
!$xmp nodes p(2,2)
!$xmp template t(1000,1000)
!$xmp template t1(2*1000,2)
!$xmp distribute t(block,block) onto p
!$xmp distribute t1(block,block) onto p
!$xmp align a(i,j) with t(i,j)
!$xmp align ipiv(i,j) with t1(i,j)
!$xmp align b(i) with t(i,*)
      ...
      integer i,j,ictxt
      integer m=1000,n=1000,nprow=2,npcol=2
      integer icontxt=-1,iwhat=0
      integer nrhs=1,ia=1,ja=1,ib=1,jb=1,info
      character*1 order
      ...
      order="C"
      ...
      call blacs_get(icontxt,iwhat,ictxt)
      call blacs_gridinit(ictxt,order,nprow,npcol)
      ...
!$xmp loop (i,j) on t(i,j)
      do j=1,n
         do i=1,m
            a(i,j) = ...
         end do
      end do
      ...
!$xmp loop on t(i,*)
      do i=1,m
         b(i)= ...
      end do
      ...
      call ixmp_pdgesv(n,nrhs,a,ia,ja,xmp_desc_of(a),ipiv,
     *                 b,ib,jb,xmp_desc_of(b),ictxt,info)
```

```
      ...
      call blacs_gridexit(ictxt)
      ...
      stop
      end
```

Appendix C

Memory-layout Model

In this chapter, the memory-layout model of global data in the Omni XcalableMP compiler (`http://www.xcalablemp.org/download.html`) is presented for reference.

According to the XcalableMP specification, a global array is distributed onto a node array according to the data-mapping directives, and as a result, a node owns a set of elements.

On each node, all and only the elements of the global array that it owns are gathered to form the local array having the same rank as the global. For each axis of the global data, all and only the indices that the node owns are packed to the axis of the local array so that the sequence can be maintained, with the shadow area, if any, added at the lower and/or upper bound of the axis.

Eventually, the local array is stored in memory on each node according to the rule for storing arrays in the base language, that is, in row-major order in XMP/Fortran and in column-major order in XMP/C.

Note that owing to the model above, the memory usage may be non-uniform among the nodes.

Example

XcalableMP Fortran

```
!$xmp nodes p(4,4)
!$xmp template t(64,64)
!$xmp distribute t(block,block) onto p

      real a(64,64)
!$xmp align a(i,j) with t(i,j)
!$xmp shadow a(1,1)
```

The array `a` is distributed by a format of `(block,block)` onto a two-dimensional node array `p`, and each node owns a local array including a shadow area. Then, the local array is stored in memory on each node, as shown in Figure C.1.

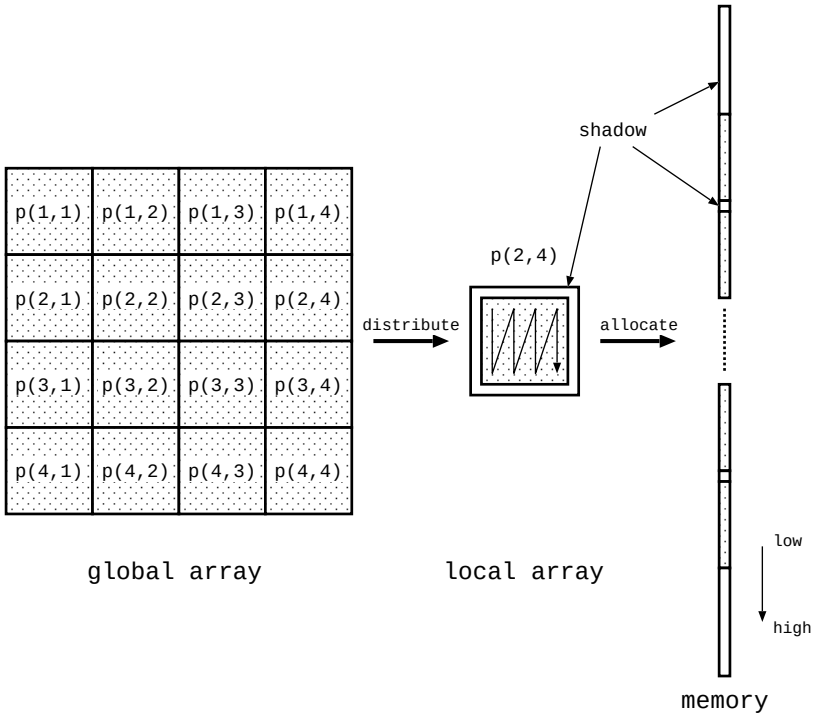

Figure C.1: Example of memory layout in the Omni XcalableMP compiler.

Appendix D

XcalableMP I/O

D.1 Categorization of I/O

XcalableMP has three kinds of I/O.

D.1.1 Local I/O

Local I/O is a method that is employed to use I/O statements and standard I/O functions in the base languages, where I/O statements and functions are used without any directives.

I/O statements (in XcalableMP Fortran) and I/O functions (in XcalableMP C) are executed locally similar to other execution statements. It depends on the system which nodes can handle the I/O statements and functions.

Local I/O can read a file written by the base language, and vice versa.

[F] The name of a global array in the I/O list describes the entire area of the array located in each node.

An array element of a global array can be referred to as an I/O item only in the node where it is located.

[F] No array section of a global array can be referred to as an I/O item.

D.1.2 Master I/O[F]

Master I/O is input and output for the file that corresponds to an executing node set. Master I/O is a collective execution.

In master I/O, a global data object is input and output as if it was executed only by a master node, which represents the executing node set, through its local copy of the data.

The master node is chosen by the system arbitrarily from among the executing node set, and is unique to the executing node set during execution of the program.

Master I/O is provided in the form of directives of XcalableMP Fortran.

A global array as an I/O item is accessed in the sequential order of array elements. When a local variable is read from a file, the value is copied to all nodes of the executing node set. When a local variable or an expression is written to a file, only the value of the data on the master node is written.

Master I/O can read a file written in the base language, and vice versa.

D.1.3 Global I/O

Global I/O is input and output for a file that corresponds to an executing node set. Some executions of the global I/O are collective and the others are independent. In a large system with

Table D.1: Global I/O.

	independent/collective	access method
Collective I/O	collective	sequential access
Atomic I/O	independent	sequential access
Direct I/O	independent	direct access

many nodes, the global I/O can be expected to have higher speed and less memory consumption execution than master I/O.

[F] It is provided in the form of directives for some of I/O statements, such as OPEN, CLOSE, READ, and WRITE statements.

[C] It is provided in the form of service functions and an include file.

Global I/O can handle only unformatted (binary) files. In XcalableMP Fortran, implied DO loops and some specifiers cannot be used. In XcalableMP C, formatted I/O libraries, including fprintf() and fscanf(), are not provided.

Global I/O can read a file written in MPI-IO, and vice versa.

[F] File formats are not compatible between XcalableMP Fortran and the base language because global I/O does not generate or access file headers and footers that are specific to the base language.

There are three kinds of global I/O, as shown in Table D.1. **Collective** global I/O is for collective execution and sequential file access. It handles global data in a sequential order, similar to master I/O. **Atomic** global I/O is for independent execution and sequential file access. Executing nodes share file positioning of the global I/O file and execute each I/O statement and library call mutually. **Direct** global I/O is for independent execution and direct file access. Each executing node has its own file positioning and accesses a shared file independently.

Restriction

- The name of a global array may not be declared in a namelist group. That is, NAMELIST I/O is not allowed for global arrays.

Advice to programmers

Local I/O is useful for debugging that is focused on a node because local I/O is executed on each node individually.

Master I/O is a directive extension, where the execution result matches that of the base language, ignoring directive lines.

Global I/O aims for highly parallel I/O using thousands of nodes. It is limited to binary files, and it avoids the extreme concentration of computational load and memory consumption to specific nodes using MPI-IO or other parallel I/O techniques.

D.2 File Connection

A file is connected to a unit in XcalableMP Fortran and to a file handler in XcalableMP C. This operation is called **file connection**. Local I/O connects a file to each node independently. Master I/O and global I/O connect a file to an executing node set collectively.

There are two ways of specifying file connections, dynamic connection and preconnection. Dynamic connection connects a file during the execution of the program. Preconnection connects a file at the beginning of execution of the program, and it can therefore execute I/O statements

and functions without the prior execution of an OPEN statement or a function call to open the file.

D.2.1 File Connection in Local I/O

The language processor of the base language connects the file to each node. It is implementation-defined which nodes can access the standard input, output, and error files. The behavior of the accesses to files having the same name on multiple nodes is also implementation-defined. The only primary node can access the standard input, output, and error files.

D.2.2 [F] File Connection in Master I/O

An OPEN statement that is specified with a master I/O directive connects a file to the executing node set. When a master I/O file is connected by a READ statement or a WRITE statement without encountering any OPEN statement, the name and attributes of the file depend on the language system of the base language. Disconnection from a master I/O file is executed by a CLOSE statement or by the termination of the program.

The dynamic connection must be executed collectively by all nodes sharing the file with the same unit number. Two executing node sets may employ the same unit number only if they have no common node.

The standard input, output, and error files are preconnected to the entire node set. Therefore, master I/O executed on the entire node set is always allowed without OPEN or CLOSE statements.

D.2.3 File Connection in Global I/O

The dynamic connection of global I/O is a collective execution, and is valid for the executing node set. Global I/O files cannot be preconnected.

[F]

An OPEN statement that is specified with a global I/O directive connects a file to the executing node set. Disconnection from a global I/O file is executed by a CLOSE statement or by the termination of the program.

The dynamic connection must be executed collectively by all nodes sharing the file with the same unit number. Two executing node sets may employ the same unit number only if they have no common node.

[C]

A library function to open a global I/O file connects the file to the executing node set. Disconnection from a global I/O file is executed by a library function to close the file or terminate the program.

D.3 Master I/O

A master I/O construct executes data transfer between a file and an executing node set via a master node of the executing node set. For a global array, the virtual sequential order of the array elements is visible.

D.3.1 `master_io` Construct

Syntax

[F] `!$xmp master_io`
io-statement

[F] `!$xmp master_io` begin
io-statement
...
`!$xmp master_io` end
where *io-statement* is one of:

- OPEN statement
- CLOSE statement
- READ statement
- WRITE statement
- PRINT statement
- BACKSPACE statement
- ENDFILE statement
- REWIND statement
- INQUIRE statement

Restriction

- The following items, including a global array or a subobject of a global array, must not appear in an input item or output item.
  - A substring-range
  - A section-subscript
  - An expression including operators
  - An *io-implied-do-control*
- An I/O statement specified with a master I/O directive must be executed collectively on the node set that is connected to the file.
- Internal file I/O is not permitted to be a master I/O.

Description

An I/O statement that is specified with a master I/O directive accesses a file whose format is the same as that of the base language. The access, including connection, disconnection, input and output, file positioning, and inquiry, is collective, and must be executed on the same node set as the one on which the file was connected.

A master node, which is a unique node to an executing node set, is chosen by the language system. Master I/O works as if all file accesses were executed only on the master node.

The operations for I/O items are summarized in Table D.2.

Table D.2: Operations for I/O.

I/O item		operation
input item	name of global array	The data elements that are read from the file in the sequential order of array elements are distributed onto the global array on the node set. The file positioning increases according to the size of data.
	array element of global array	The data element that is read from the file is copied to the element of the global array on the node to which it is mapped. The file positioning increases according to the size of data.
	local variable	The data element that is read from the file is replicated to the local variables on all nodes of the executing node set. The file positioning increases according to the size of data.
	implied DO loop	For each input item, repeat the above operation.
output item	name of global array	The data elements of the global array are collected and are written to the file in the sequential order of array elements. The file positioning increases according to the size of data.
	array element of global array	The element of the global array is written to the file. A file position increases according to the size of data.
	local variable and expression	The value evaluated on the master node is written to the file. The file positioning increases according to the size of data.
	implied DO loop	For each output item, repeat the above operation.

Namelist input and output statements cannot treat global arrays. A namelist output statement writes the values on the master node to the file. In the namelist input, each item of the namelist is read from the file to the master node if it is recorded in the file. Then, all items of the namelist are replicated onto all nodes of the executing node set from the master node even if some items are not read from the file.

IOSTAT and SIZE specifiers and specifiers of the INQUIRE statement that can return values always return the same value among the executing node set.

When a condition that is specified by the ERR, END, or EOR specifier is satisfied, all nodes of the executing node set are branched together to the same statement.

Advice to implementers

It is recommended to provide such a compiler option that local I/O statements (specified without directives) are regarded as master I/O statements (specified with `master_io` directives).

D.4 [F] Global I/O

Global I/O performs unformatted data transfer, and can be expected to have a higher performance and lower memory consumption than master I/O. The file format is compatible with the

one in MPI-IO.

There are three kinds of Global I/O, namely collective I/O, atomic I/O, and direct I/O.

D.4.1 Global I/O File Operation

`global_io` construct is defined as follows.

Syntax

[F] `!$xmp global_io` [atomic / direct]
io-statement

[F] `!$xmp global_io` [atomic / direct] begin
io-statement
...
`!$xmp` end `global_io`

The first syntax is just a shorthand of the second syntax.

Restriction

I/O statements and specifiers that are available for an *io-statement* are shown in the following table. The definition of each specifier is described in the specification of the base language.

Case of `global_io` construct without a direct clause:

I/O statement	available specifiers
OPEN	UNIT, IOSTAT, FILE, STATUS, POSITION, ACTION, ACCESS, FORM
CLOSE	UNIT, IOSTAT, STATUS
READ	UNIT, IOSTAT
WRITE	UNIT, IOSTAT

Case of `global_io` construct with a direct clause:

I/O statement	available specifiers
OPEN	UNIT, IOSTAT, FILE, STATUS, RECL, ACTION, ACCESS, FORM
CLOSE	UNIT, IOSTAT, STATUS
READ	UNIT, REC, IOSTAT
WRITE	UNIT, REC, IOSTAT

The input item and output item of a data transfer statement with a `global_io` directive must be the name of a variable.

Description

Global I/O construct connects, disconnects, inputs, and outputs the global I/O file, which is compatible with MPI-IO.

The standard input, output, and error files cannot be a Global I/O file. A Global I/O file cannot preconnect to any unit or any file handler, and must be explicitly connected by the OPEN statement that is specified with a `global_io` directive.

The OPEN statement that is specified with a `global_io` directive is collective execution, and the file is shared among the executing node set. A file that has already been opened by another

OPEN statement with a `global_io` directive cannot be reopened by an OPEN statement with or without a `global_io` directive before closing it.

A global I/O file must be disconnected explicitly by a CLOSE statement that is specified with a `global_io` directive; otherwise, the result of I/O is not guaranteed. The CLOSE statement that is specified with a `global_io` directive is a collective execution, and must be executed by the same executing node set as the one where the OPEN statement is executed.

Utilizable values of the specifiers in I/O statements are shown in the following table. Definitions of the specifiers are described in the specification of the base language.

- OPEN statement

specifiers	value	default
UNIT	external file unit (scalar constant expression)	not omissible
FILE	file name (scalar CHARACTER expression)	not omissible
STATUS	'OLD', 'NEW', 'REPLACE' or 'UNKNOWN'	'UNKNOWN'
POSITION	'ASIS', 'REWIND' or 'APPEND'	'ASIS'
ACTION	'READ', 'WRITE' or 'READWRITE'	implementation-defined
RECL	the value of the record length (scalar constant expression)	not omissible
ACCESS	'SEQUENTIAL' or 'DIRECT'	'SEQUENTIAL'
FORM	'FORMATTED' or 'UNFORMATTED'	For direct access, UNFORMATTED. For sequential access, this specifier shall not be omitted.

  POSITION is available only if the directive has no direct clause. RECL is available only if the directive has a direct clause. For direct I/O, the ACCESS specifier shall appear and the value shall be evaluated to DIRECT. For collective I/O and atomic I/O, the value of the ACCESS specifier shall be evaluated to SEQUENTIAL if this specifier appears. For collective I/O and atomic I/O, the FORM specifier shall appear and the value shall be evaluated to UNFORMATTED. For direct I/O, the value of the FORM specifier shall be evaluated to UNFORMATTED if this specifier appears.

- CLOSE statement

specifiers	value	default
UNIT	external file unit (scalar constant expression)	not omissible.
STATUS	'KEEP' or 'DELETE'	'KEEP'

- READ/WRITE statement

  REC is available only if the directive has a direct clause.

- When a scalar variable of default INTEGER is specified to the IOSTAT specifier, it becomes defined with an error code after execution.

specifiers	value	default
UNIT	external file unit (scalar constant expression)	not omissible
REC	the value of the number of record (scalar constant expression)	not omissible

OPEN, CLOSE, READ, and WRITE statements that are specified with `global_io` directives without atomic or direct clauses are called collective OPEN, collective CLOSE, collective READ, and collective WRITE statements, respectively. All of these statements are called collective I/O statements.

OPEN, CLOSE, READ, and WRITE statements specified with `global_io` directives having atomic clauses are called atomic OPEN, atomic CLOSE, atomic READ, and atomic WRITE statements, respectively. All of these statements are called atomic I/O statements.

OPEN, CLOSE, READ, and WRITE statements specified with `global_io` directives with direct clauses are called direct OPEN, direct CLOSE, direct READ, and direct WRITE statements, respectively. All of these statements are called direct I/O statements.

The file connected by a collective, atomic, or direct OPEN statement can be read/written only by the same type of READ/WRITE statements. The file can be disconnected by the same type of CLOSE statement. Different types of global I/O cannot be executed together for the same file or the same unit. For example, atomic I/O statements cannot be executed for the unit connected by a collective OPEN statement.

D.4.1.1 `file_sync_all` Directive

Two data accesses cause a conflict if they access the same absolute byte displacements of the same file, and at least one is a write access. When two accesses to the same file conflict in direct or collective I/O, the following file_sync_all directive to the file must be executed.

Syntax

`!$xmp file_sync_all`([UNIT=]file-unit-number)

The file_sync_all directive is an execution directive and collective to the nodes connected to the specified file-unit-number. The execution of a file_sync_all directive first synchronizes all the nodes connected to the specified file-unit-number, and then causes all previous writes to the file by the nodes to be transferred to the storage device. If some nodes have made updates to the file, then all such updates become visible to subsequent reads of the file by the nodes.

D.4.2 Collective Global I/O Statement

Collective I/O statements read/write shared files and can handle global arrays.

All collective I/O statements execute collectively. In collective I/O, all accesses to a file, such as connection, disconnection, input, and output, must be executed on the same executing node set.

The operations for I/O items are summarized in the following table.

D.4.3 Atomic Global I/O Statement

Atomic I/O statements read/write shared files exclusively among executing nodes in arbitrary order. Because it is a nondeterministic parallel execution, the results may differ every time it is executed, even for the same program.

I/O item		operation
input item	name of global array	The values read from a file are assigned to the elements of the global array. The file positioning increases according to the size of the data.
	local variable	The values read from the file are replicated into the local array on all executing nodes. The file positioning increases according to the length of the data.
output item	name of global array	The values of a global array are written to the file in the sequential order of the array elements. The file positioning increases according to the size of the data.
	local variable, expression	The values evaluated on a node are arbitrarily selected by the language processor from the executing node set. The file positioning increases according to the size of the data.

Atomic OPEN and CLOSE statements are executed collectively, while atomic READ and WRITE statements are executed independently. A file connected by an atomic OPEN statement can be disconnected only by an atomic CLOSE statement executed on the same executing node set. Atomic READ and WRITE statements can be executed on any single node of the same executing node set.

Atomic READ and WRITE statements are exclusively executed. The unit of exclusive operation is a single READ statement or a single WRITE statement.

The initial file positioning is determined by the POSITION specifier of the atomic OPEN statement. Then, the file positioning seeks in every READ and WRITE statement according to the length of the input/output data.

D.4.4 Direct Global I/O Statement

Direct I/O statements read/write shared files by specifying the file positioning for each node.

Direct OPEN and CLOSE statements are executed collectively, while direct READ and WRITE statements are executed independently. A file connected by a direct OPEN statement can be disconnected only by a direct CLOSE statement executed on the same executing node set. Direct READ and WRITE statements can be executed on any single node of the same executing node set.

Direct READ and WRITE statements read/write local data at the file positioning specified by the REC specifier independently. The file positioning is shifted from the top of the file on the basis of the product of the specifiers RECL (of OPEN statement) and REC (of READ and WRITE statement).

In order to guarantee the order of direct I/O statements to the same file position, the file should be closed or the file_sync_all directive should be executed between these statements. Otherwise, the outcome of multiple accesses to the same file position, in which at least one is a write access, is implementation-defined.

D.5 [C] Global I/O Library

XcalableMP C provides some data types defined in the include file “xmp.h”, a set of library functions with arguments of the data types, and built-in operators to get values of the data types from names of a variable, a template, etc.

The following types are provided.

- xmp_file_t : file handle
- xmp_rang_t : descriptor of array section

The following library functions are provided. Collective function names end with _all.

- global I/O file operation
  - xmp_fopen_all : file open
  - xmp_fclose_all : file close
  - xmp_fseek : setting (individual) file pointer
  - xmp_fseek_shared_all : setting shared file pointer
  - xmp_ftell : displacement of (individual) file pointer
  - xmp_ftell_shared : displacement of shared file pointer
  - xmp_file_sync_all : file synchronization
- collective I/O
  - xmp_file_set_view_all : setting file view
  - xmp_file_clear_view_all : initializing file view
  - xmp_fread_all : collective read of local data
  - xmp_fwrite_all : collective write of local data
  - xmp_fread_darray_all : collective read of global data
  - xmp_fwrite_darray_all : collective write of global data
- atomic I/O
  - xmp_fread_shared : atomic read
  - xmp_fwrite_shared : atomic write
- direct I/O
  - xmp_fread : direct read
  - xmp_fwrite : direct write

Data type

The following data types are defined in include file xmp_io.h.

xmp_file_t A file handler. It is connected to a file when the file is opened. It has a shared file pointer and an individual file pointer to point where data should be read/written in the file.

A shared file pointer is a shared resource among all nodes of the node set that has opened the file. Atomic I/O uses a shared file pointer. An (individual) file pointer is an individual resource on each node. Collective I/O and direct I/O use individual file pointers.

These two file pointers are managed in the structure xmp_file_t, and can be controlled and referenced only through the provided library functions.

xmp_range_t Descriptor of array section, including lower bound, upper bound, and stride for each dimension. Functions for operating the descriptor are shown in the following table. The xmp_allocate_range() function is used to allocate memory. The xmp_set_range() function is used to set ranges of an array section. The xmp_free_range() function releases the memory for the descriptor.

function name	**xmp_range_t *xmp_allocate_range(n_dim)**	
argument	int n_dim	the number of dimensions
return value	xmp_range_t*	descriptor of array section. NULL is returned when a program abends.

function name	**void xmp_set_range(rp, i_dim, lb, length, step)**	
argument	xmp_range_t *rp	descriptor
	int i_dim	target dimension
	int lb	lower bound of array section in the dimension i_dim
	int length	length of array section in the dimension i_dim
	int step	stride of array section in the dimension i_dim

function name	void xmp_free_range(rp)	
argument	xmp_range_t *rp	descriptor of array section.

D.5.1 Global I/O File Operation

D.5.1.1 xmp_fopen_all

xmp_fopen_all opens a global I/O file. Collective execution.

function name	xmp_file_t *xmp_fopen_all(fname, amode)	
argument	const char *fname	file name
	const char *amode	equivalent to fopen of POSIX. combination of "rwa+"
return value	xmp_file_t*	file structure. NULL is returned when a program abend.

File view is initialized, where file view is based on the MPI-IO file view mechanism. The value of shared and individual file pointers depends on the value of amode.

amode	intended purpose
r	Open for reading only. File pointer points to the beginning of the file.
r+	Open an existing file for update (reading and writing). File pointer points to the beginning of the file.
w	Create for writing. If a file having that name already exists, it will be overwritten. File pointer points to the beginning of the file.
w+	Create a new file for update (reading and writing). If a file having that name already exists, it will be overwritten. File pointer points to the beginning of the file.
a	Append; open for writing at end-of-file or create for writing if the file does not exist. File pointer points to the end of the file.
a+	Open for append; open (or create if the file does not exist) for update at the end of the file. File pointer points to the beginning of the file.

D.5.1.2 xmp_fclose_all

xmp_fclose_all closes a global I/O file. Collective execution.

function name	int *xmp_fclose_all(fh)	
argument	xmp_file_t *fh	file structure
return value	int	0: normal termination 1: abnormal termination. fh is NULL. 2: abnormal termination. error in MPI_File_close.

D.5.1.3 xmp_fseek

xmp_fseek sets the individual file pointer in the file structure. Independent execution.

function name	int xmp_fseek(fh, offset, whence)	
argument	xmp_file_t *fh	file structure
	long long offset	displacement of current file view from position of whence
	int whence	choose file position SEEK_SET: the beginning of the file SEEK_CUR: current position SEEK_END: the end of the file
return value	int	0: normal termination an integer other than 0: abnormal termination

D.5.1.4 xmp_fseek_shared

xmp_fseek_shared sets the shared file pointer in the file structure. Independent execution.

function name	int xmp_fseek_shared(fh, offset, whence)	
argument	xmp_file_t *fh	file structure
	long long offset	displacement of current file view from position of whence
	int whence	choose file position SEEK_SET: the beginning of the file SEEK_CUR: current position SEEK_END: the end of the file
return value	int	0: normal termination an integer other than 0: abnormal termination

D.5.1.5 xmp_ftell

xmp_ftell returns the position of the individual file pointer in the file structure. Independent execution.

function name	long long xmp_ftell(fh)	
argument	xmp_file_t *fh	file structure
return value	long long	Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest-numbered unused file descriptor. Otherwise, a negative number shall be returned.

D.5.1.6 xmp_ftell_shared

xmp_ftell_shared returns the position of the shared file pointer in the file structure. Independent execution.

function name	long long xmp_ftell_shared(fh)	
argument	xmp_file_t *fh	file structure
return value	long long	Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, negative number shall be returned.

D.5.1.7 xmp_file_sync_all

xmp_file_sync_all guarantees completion of access to the file from nodes sharing the file. Two data accesses conflict if they access the same absolute byte displacements of the same file, and at least one is a write access. When two accesses A1 and A2 to the same file conflict in direct or collective I/O, an xmp_file_sync_all to the file must be invoked between A1 and A2; otherwise, the outcome of the accesses is undefined. Collective execution.

function name	int xmp_file_sync_all(fh)	
argument	xmp_file_t *fh	file structure
return value	int	0: normal termination an integer other than 0: abnormal termination

D.5.2 Collective Global I/O Functions

Collective I/O is executed collectively, but using the individual pointer. It reads/writes data from the position of the individual file pointer and moves the position forward by the length of the data.

Before the file access, a file view is often specified. A file view, like a window to the file, spans the positions corresponding to the array elements that are owned by each node. For more details of file view, refer to the MPI 2.0 specification.

D.5.2.1 xmp_file_set_view_all

xmp_file_set_view_all sets a file view to the file. Collective execution.

function name	int xmp_file_set_view_all(fh, disp, desc, rp)	
argument	xmp_file_t *fh	file structure
	long long disp	displacement from the beginning of the file.
	xmp_desc_t desc	descriptor
	xmp_range_t *rp	range descriptor
return value	int	0: normal termination an integer other than 0: abnormal termination

The file view of distributed *desc* limited to range *rp* is set into file structure fh.

D.5.2.2 xmp_file_clear_view_all

xmp_file_clear_view_all clears the file view. Collective execution.

The positions of the shared and individual file pointers are set to disp, and the elemental data type and the file type are set to MPI_BYTE.

function name	**int xmp_file_clear_view_all(fh, disp)**	
argument	xmp_file_t *fh	file structure
	long long disp	displacement from the beginning of the file.
return value	int	0: normal termination an integer other than 0: abnormal termination

D.5.2.3 xmp_fread_all

xmp_fread_all reads the same data from the position of the shared file pointer onto all of the executing nodes. Collective execution.

function name	**ssize_t xmp_fread_all(fh, buffer, size, count)**	
argument	xmp_file_t *fh	file structure
	void *buffer	beginning address of read variables
	size_t size	the size of a read data element
	size_t count	the number of read data elements
return value	ssize_t	Upon successful completion, return the size of read data. Otherwise, negative number shall be returned.

D.5.2.4 xmp_fwrite_all

xmp_fwrite_all writes individual data on all of the executing nodes to the position of the shared file pointer. Collective execution.

It is assumed that the file view is set in advance. Each node writes its data into its own file view.

function name	**ssize_t xmp_fwrite_all(fh, buffer, size, count)**	
argument	xmp_file_t *fh	file structure
	void *buffer	beginning address of written variables
	size_t size	the size of a written data element
	size_t count	the number of written data elements
return value	ssize_t	Upon successful completion, return the size of written data. Otherwise, negative number shall be returned.

D.5.2.5 xmp_fread_darray_all

xmp_fread_darray_all reads data cooperatively to the global array from the position of the shared file pointer.

Data is read from the file to distributed *desc* limited to range *rp*.

function name	ssize_t xmp_fread_darray_all(fh, desc, rp)	
argument	xmp_file_t *fh	file structure
	xmp_desc_t desc	descriptor
	xmp_range_t *rp	range descriptor
return value	ssize_t	Upon successful completion, return the size of read data. Otherwise, negative number shall be returned.

D.5.2.6 xmp_fwrite_darray_all

xmp_fwrite_darray_all writes data cooperatively from the global array to the position of the shared file pointer.

function name	ssize_t xmp_fwrite_darray_all(fh, desc, rp)	
argument	xmp_file_t *fh	file structure
	xmp_desc_t desc	descriptor
	xmp_range_t *rp	range descriptor
return value	ssize_t	Upon successful completion, return the size of read data. Otherwise, negative number shall be returned.

Data is written from distributed *desc* limited to range *rp* to the file.

D.5.3 Atomic Global I/O Functions

Atomic I/O is executed independently, but using the shared pointer. It exclusively reads/writes local data from the position of the shared file pointer, and moves the position forward by the length of the data.

Before atomic I/O is executed, the file view must be cleared.

[Rationale]

Although the file views must be the same on all processes in order to use the shared file pointer, the xmp_file_set_view_all function may set different file views for all nodes. Thus, before atomic I/O is used, the file view must be cleared.

D.5.3.1 xmp_fread_shared

xmp_fread_shared exclusively reads local data from the position of the shared file pointer, and moves the position forward by the length of the data. Independent execution.

function name	ssize_t xmp_fread_shared(fh, buffer, size, count)	
argument	xmp_file_t *fh	file structure
	void *buffer	beginning address of read variables
	size_t size	the size of a read data element
	size_t count	the number of read data elements
return value	ssize_t	Upon successful completion, return the size of read data. Otherwise, negative number shall be returned.

D.5.3.2 xmp_fwrite_shared

xmp_fwrite_shared exclusively writes local data to the position of the shared file pointer and moves the position forward by the length of the data. Independent execution.

function name	ssize_t xmp_fwrite_shared(fh, buffer, size, count)	
argument	xmp_file_t *fh	file structure
	void *buffer	beginning address of written variables
	size_t size	the size of a written data element
	size_t count	the number of written data elements
return value	ssize_t	Upon successful completion, return the size of written data. Otherwise, negative number shall be returned.

D.5.4 Direct Global I/O Functions

Direct I/O is executed independently and uses the individual pointer. It individually reads/writes local data from the position of the individual file pointer, and moves the position forward by the length of the data, considering the file view.

In order to guarantee the order by xmp_fread and xmp_fwrite functions to the same file position, the file should be closed or the xmp_file_sync_all function should be executed between these functions. Otherwise, the outcome of multiple accesses to the same file position, in which at least one is the xmp_fwrite function, is implementation dependent.

Advice to programmers

Function xmp_fseek is useful for setting the individual file pointer. It is not recommended to use it together with the file view because of its complexity.

D.5.4.1 xmp_fread

xmp_fread reads data from the position of the individual file pointer and moves the position forward by the length of the data. Independent execution.

function name	ssize_t xmp_fread(fh, buffer, size, count)	
argument	xmp_file_t *fh	file structure
	void *buffer	beginning address of read variables
	size_t size	the size of a read data element
	size_t count	the number of read data elements
return value	ssize_t	Upon successful completion, return the size of read data. Otherwise, negative number shall be returned.

D.5.4.2 xmp_fwrite

xmp_fwrite writes data to the position of the individual file pointer and moves the position forward jby the length of the data. Independent execution.

function name	**ssize_t xmp_fwrite(fh, buffer, size, count)**	
argument	xmp_file_t *fh	file structure
	void *buffer	beginning address of written variables
	size_t size	the size of a written data element
	size_t count	the number of written data elements
return value	ssize_t	Upon successful completion, return the size of written data. Otherwise, negative number shall be returned.

Appendix E

Memory Consistency Model

This chapter explains the memory consistency model that is adopted by XcalableMP.

Memory consistency models have specified rules regarding multiple data accesses to memory. Because XcalableMP is an extension of the base languages, and its memory consistency model is defined as an extension to them, that is, XcalableMP follows all of the rules that are adopted by base languages.

In addition, XcalableMP introduces some rules about global view. In global view, global communication constructs are used to access distributed data. Furthermore, distributed data can be accessed by designating data in local view. Conversely, non-distributed data can be accessed by designating distributed data using global communication constructs in global view. These are not considered under the memory consistency models of the base language because global view is a new concept that was introduced by XcalableMP.

Please recall that global communication constructs are collective as described in Section 2.8.

E.1 Execution Traces

This section explains execution traces that are enabled by the Xcalable memory consistency model.

First, instructions are defined as

$$i := \texttt{xmp\_syn} \mid \texttt{xmp\_asyn}(\textit{async-id}) \mid \texttt{wait\_async}(\textit{async-id}) \mid \texttt{f\_stmt}$$

where `xmp_syn` denotes a global communication construct with no `async` clause, `xmp_asyn`(*async-id*) denotes a global communication construct with the clause `async`(*async-id*), and `f_stmt` is a statement.

Next, operations are defined as

$$o := \texttt{Fetch}^j\ i \mid \texttt{Execute}^j\ i \mid \texttt{Reflect}^j\ i$$

where j is a positive integer.

Operation $\texttt{Fetch}^j\ i$ denotes that instruction i is fetched j times. The integer j is incremented whenever a loop is exited. The instructions that are called multiple times in loops are identified by js. Operation $\texttt{Execute}^j\ i$ denotes that instruction i is executed, while operation $\texttt{Reflect}^j\ i$ denotes that the effect of instruction i is saved to physical memories.

Finally, the memory consistency model defines constraints written by a partial order $\leq$ on operations as described below. Execution traces are defined as sequences of operations that follow the order. In the following, $o_1 < o_2$ denotes $o_1 \leq o_2$ and $o_1 \not\equiv o_2$. In addition, $o_1 < o_2 < o_3$ denotes $o_1 < o_2$ and $o_2 < o_3$.

$\texttt{Fetch}^{j_1}\ i_1 < \texttt{Fetch}^{j_2}\ i_2$ implies $\texttt{Execute}^{j_1}\ i_1 < \texttt{Execute}^{j_2}\ i_2$ (i)

$\texttt{Execute}^{j_1}\ \texttt{xmp\_syn} < \texttt{Execute}^{j_2}\ i_2$ implies $\texttt{Reflect}^{j_1}\ \texttt{xmp\_syn} < \texttt{Execute}^{j_2}\ i_2$ (ii)

$\texttt{Execute}^{j_1}\ \texttt{xmp\_asyn}(\textit{async-id}) < \texttt{Execute}^{j_3}\ \texttt{wait\_async}(\textit{async-id}) < \texttt{Execute}^{j_2}\ i_2$ implies
$\texttt{Reflect}^{j_1}\ \texttt{xmp\_asyn}(\textit{async-id}) < \texttt{Execute}^{j_2}\ i_2$ (iii)

Figure E.1: Constraints that are required by the XcalableMP memory consistency model.

E.1.1 Common Constraints

In this subsection, we explain some constraints that are common to both synchronous and asynchronous communications.

In the XcalableMP memory consistency model, instructions are executed in the order in which they are fetched. This is represented by i in Figure E.1.

E.1.2 Constraints for Synchronous Communications

The constructs `reflect`, `gmove` (and its subsequent assignment statement), `reduction`, and `bcast` are synchronous if `async` is not specified. This means that executions of these constructs guarantee the completion of data synchronization. That is, global communication constructs read data that are written by previously executed statements, and their subsequent statements and global communication constructs read data that are written by global communication constructs. This is given by ii in Figure E.1

For example, in the following code, the assignment statement `g(:)=h(:)` is guaranteed to be completed before the second `gmove` construct is executed. Therefore, the value of `g(i)` must be `i` when the assignment statement `x(:)=g(6:10)` is executed.

Finally, the value of `x(i)` on `p(1)` should be `i+5`.

XcalableMP Fortran

```
!$xmp nodes p(2)
!$xmp template t(10)
!$xmp distribute (block) onto p :: t
      integer :: g(10), h(10)
!$xmp align (i) with t(i) :: g, h
      integer x(5)

!$xmp loop on t(i)
      do i=1,10
      h(i)=i
      end do

!$xmp gmove
      g(:)=h(:)
!$xmp gmove
      x(:)=g(6:10)
```

E.1.3 Constraints for Asynchronous Communications

The constructs `reflect`, `gmove` (and its following assignment statement), `reduction`, and `bcast` are asynchronous if `async`s are specified. Completions of data read and written by these global

communication constructs are not guaranteed until `wait_async`s are executed. This is represented by iii in Figure E.1.

For example, in the following code, the assignment statement `g(:)=h(:)` may not be completed before the second `gmove` construct is executed as the first `gmove` construct has `async` clause. Therefore, the value of `g(i)` is not guaranteed to be `i+5`. Of course, the value of `x(i)` on `p(1)` is not guaranteed to be `i+5`.

XcalableMP Fortran

```
!$xmp nodes p(2)
!$xmp template t(10)
!$xmp distribute (block) onto p :: t
      integer :: g(10), h(10)
!$xmp align (i) with t(i) :: g, h
      integer x(5)

!$xmp loop on t(i)
      do i=1,10
      h(i)=i
      end do

!$xmp gmove async(1)
      g(:)=h(:)
!$xmp gmove
      x(:)=g(6:10)
!$xmp wait_async(1)
```

The `wait_async(`*`async-id`*`)` guarantees the completion of a global communication construct that has *async-id*. Therefore, the value of `x(i)` is not guaranteed to be `i+5` in the following program:

XcalableMP Fortran

```
!$xmp nodes p(2)
!$xmp template t(10)
!$xmp distribute (block) onto p :: t
      integer :: g(10), h(10)
!$xmp align (i) with t(i) :: g, h
      integer x(5)

!$xmp loop on t(i)
      do i=1,10
      h(i)=i
      end do

!$xmp gmove async(1)
      g(:)=h(:)
!$xmp wait_async(1)
!$xmp gmove
      x(:)=g(6:10)
```

Assignment statements in local view and `gmove` constructs in global view may race. The value of `x(5)` is not guaranteed to be `6`, and may be `10` in the following program:

XcalableMP Fortran

```
!$xmp nodes p(2)
!$xmp template t(10)
!$xmp distribute (block) onto p :: t
      integer :: g(10), h(10)
!$xmp align (i) with t(i) :: g, h
      integer x(5)

      integer l(5), m(5)
!$xmp local_alias l => g
!$xmp local_alias m => h

!$xmp loop on t(i)
      do i=1,10
      h(i)=i
      end do

!$xmp gmove async(1)
      g(:)=h(:)
      l(5)=6
!$xmp wait_async(1)
      x(5)=l(5)
```

By avoiding the race, the value of `x(5)` is guaranteed to be `6` as follows:

XcalableMP Fortran

```
!$xmp nodes p(2)
!$xmp template t(10)
!$xmp distribute (block) onto p :: t
      integer :: g(10), h(10)
!$xmp align (i) with t(i) :: g, h
      integer x(5)
      integer l(5), m(5)
!$xmp local_alias l => g
!$xmp local_alias m => h

!$xmp loop on t(i)
      do i=1,10
      h(i)=i
      end do

!$xmp gmove async(1)
      g(:)=h(:)
!$xmp wait_async(1)
      l(5)=6
      x(5)=l(5)
```

Please note that function calls have no synchronization at its entrance/exit. In the following program, the value of `x(5)` is not guaranteed to be `6`:

XcalableMP Fortran

```
!$xmp nodes p(2)
!$xmp template t(10)
```

```
!$xmp distribute (block) onto p :: t
      integer :: g(10), h(10)
!$xmp align (i) with t(i) :: g, h
      integer x(5)
      integer l(5), m(5)
!$xmp local_alias l => g
!$xmp local_alias m => h

!$xmp loop on t(i)
      do i=1,10
      h(i)=i
      end do

!$xmp gmove async(1)
      call sub(g,h)
      l(5)=6
!$xmp wait_async(1)
      x(5)=l(5)
```

Appendix F

Sample Programs

Example 1

XcalableMP C

```
/*
 *  A parallel explicit solver of Laplace equation in \XMP
 */
#pragma xmp nodes p(NPROCS)
#pragma xmp template t(1:N)
#pragma xmp distribute t(block) onto p

double u[XSIZE+2][YSIZE+2],
       uu[XSIZE+2][YSIZE+2];
#pragma xmp align u[i][*] to t(i)
#pragma xmp align uu[i][*] to t(i)
#pragma xmp shadow uu[1:1][0:0]

lap_main()
{
 int x,y,k;
 double sum;
for(k = 0; k < NITER; k++){
        /* old <- new */
#pragma xmp loop on t(x)
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            uu[x][y] = u[x][y];
#pragma xmp reflect (uu)
#pragma xmp loop on t(x)
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            u[x][y] = (uu[x-1][y] + uu[x+1][y] +
                  uu[x][y-1] + uu[x][y+1])/4.0;
    }

    sum = 0.0;
#pragma xmp loop on t[x] reduction(+:sum)
    for(x = 1; x <= XSIZE; x++)
        for(y = 1; y <= YSIZE; y++)
```

```
          sum += (uu[x][y]-u[x][y]);
#pragma xmp task on p(1)
    printf("sum = %g\n",sum);
}
```

Example 2

XcalableMP C

```
/*
 * Linpack in XcalableMP (Gaussian elimination with partial pivoting)
 *    1D distribution version
 */
#pragma xmp nodes p(*)
#pragma xmp template t(0:LDA-1)
#pragma xmp distribute t(cyclic) onto p

double pvt_v[N];  // local

/*     gaussian elimination with partial pivoting        */
dgefa(double a[n][LDA],int lda, int n,int ipvt,int *info)
#pragma xmp align a[:][i] with t(i)
{
    REAL t;
    int idamax(),j,k,kp1,l,nm1,i;
    REAL x_pvt;

    nm1 = n - 1;
    for (k = 0; k < nm1; k++) {
      kp1 = k + 1;
      /* find l = pivot index        */
      l = A_idamax(k,n-k,a[k]);
      ipvt[k] = l;

      /* if (a[k][l] != ZERO) */
#ifdef XMP
#pragma xmp gmove
      pvt_v[k:n-k] = a[l][k:n-k];
#else
      for(i = k; i < n; i++) pvt_v[i] = a[i][l];
#endif

      /* interchange if necessary */
      if (l != k){
#ifdef XMP
#pragm xmp gmove
        a[l][:] = a[k][:];
#pramga xmp gmove
        a[k][:] = pvt_v[:];
#else
        for(i = k; i< n; i++) a[i][l] = a[i][k];
        for(i = k; i< n; i++) a[i][k] = pvt_v[i];
```

```
#endif
      }
      /* compute multipliers */
      t = -ONE/pvt_v[k];
      A_dscal(k+1, n-(k+1),t,a[k]);

      /* row elimination with column indexing */
      for (j = kp1; j < n; j++) {
        t = pvt_v[j];
        A_daxpy(k+1,n-(k+1),t,a[k],a[j]);
      }
    }
    ipvt[n-1] = n-1;
}

dgesl(double a[n][LDA],int lda,int n,int pvt[n],double b,int job)
#pragma xmp align a[:][i] with t(i)
#pragma xmp align b[i] with t(i)
{
    REAL t;
    int k,kb,l,nm1;

    nm1 = n - 1;
    /* job = 0 , solve  a * x = b,  first solve  l*y = b  */
    for (k = 0; k < nm1; k++) {
        l = ipvt[k];
#pragma xmp gmove
        t = b[l];
        if (l != k){
#pragma xmp gmove
            b[l] = b[k];
#pragma xmp gmove
            b[k] = t;
        }
        A_daxpy(k+1,n-(k+1),t,a[k],b);
    }

    /* now solve u*x = y */
    for (kb = 0; kb < n; kb++) {
        k = n - (kb + 1);
#pragma xmp task on t(k)
{
        b[k] = b[k]/a[k][k];
        t = -b[k];
}
#pragma xmp bcast (t) from t(k)
        A_daxpy(0,k,t,a[k],b);
    }
}
```

```
/*
* distributed array based routine
*/
A_daxpy(int b,int n,double da,double dx[n],double dy[n])
#pragma xmp align dx[i] with t(i)
#pragma xmp align dy[i] with t(i)
{
    int i,ix,iy,m,mp1;
    if(n <= 0) return;
    if(da == ZERO) return;
    /* code for both increments equal to 1 */
#pragma xmp loop on t(b+i)
    for (i = 0;i < n; i++) {
        dy[b+i] = dy[b+i] + da*dx[b+i];
    }
}

int A_idamax(int b,int n,double dx[n])
#pragma xmp align dx[i] with t(i)
{
  double dmax, g_dmax;
    int i, ix, itemp;
    if(n == 1) return(0);

    /* code for increment equal to 1 */
    itemp = 0;
    dmax = 0.0;
#pragma xmp loop on t(i) reduction(lastmax:dmax/itemp/)
    for (i = b; i < n; i++) {
      if(fabs((double)dx[i]) > dmax) {
        itemp = i;
        dmax = fabs((double)dx[i]);
      }
    }
    return (itemp);
}

A_dscal(int b,int n,double da,double dx[n])
#pragma xmp align dx[i] with t(i)
#pragma xmp align dy[i] with t(i)
{
    int i;
    if(n <= 0)return;

    /* code for increment equal to 1 */
#pragma xmp loop on t(i)
    for (i = b; i < n; i++)
      dx[i] = da*dx[i];
}
```

Appendix G

DRAFT: Coarray Features

This chapter is a proposal document to be added before Section 5.

For the local-view programming, XcalableMP supports the coarray features as a part of the language specifications. XcalableMP Fortran contains all coarray features defined in the standard Fortran 2008 (ISO/IEC 1539-1:2010) with few incompatibility described in Section G.8 and includes some important intrinsic procedures defined in the standard Fortran 2015. Also XcalableMP C contains the coarray features which was designed based on the ones of XcalableMP Fortran.

G.1 Introduction for Coarrays

Image and image index

The local-view programming model is a Single Program Multiple Data (SPMD) model. Each replication of the program is called an **image**. Every image has a different **image index**, which is an integer number between one and the number of images. The number of images is not determined until the program execution.

In XcalableMP, a virtual array of the whole images with any number of dimensions is called an **image array**. Each array element of an image array is corresponding to an image index in the array element order of Fortran. The extent of the final (outermost) dimension is not determined until the program execution because it depends on the number of images.

The images are mapped one-by-one to the execution nodes. The correspondence between images and nodes is defined later. Inquire functions about the image index and the number of images are described in Section G.7.

Coarray

A **coarray** is an object that has a corresponding image array. Unlike a usual object (non-coarray), a coarray is allowed to be referred from other images. Each image has its own coarrays and can reference and define coarrays on all images each other.

The shape of the image array corresponding to a coarray is called a **coshape** of the coarray. A coshape is specified with a *coarray-spec* in the declaration of the coarray variable or the coarray pointer (described later). The number of dimensions of a coshape is called a **corank**. For each dimension of a coshape, the lower and upper bounds and the extent are called **lower and upper cobounds** and **coextent**, respectively.

[F] A *coarray-spec* can appear in a type declaration statement and in a component definition statement. Entities declared with a *coarray-spec* are categorized into three:

- A **static coarray** is a coarray that is not a dummy argument and is non-allocatable. It must have the SAVE attribute explicitly or implicitly.
- A **dummy coarray** is a coarray that is a dummy argument and is non-allocatable. The actual argument corresponding to a dummy coarray must be a static coarray, a dummy coarray, an allocatable coarray, or a subobject of them.
- An **Allocatable coarray** is a coarray that is allocatable. An ultimate component[1] of a non-coarray structure can also be an allocatable coarray.

[C] A *coarray-spec* can appear in a *declaration* and in a *parameter-declaration*. Entities declared with a *coarray-spec* are categorized into three:

- A **static coarray** is a coarray that is not a dummy argument and is of a basic, structure or array type. If it is an array, the array element must be of a basic, structure or array type. It must have the `static` or `extern` storage class.
- A **dummy coarray** is a coarray that is a dummy argument and is of a basic, structure or array type. If it is an array, the array element must be of a basic, structure or array type. The actual argument corresponding to a dummy coarray must be the name of a static corray, dummy coarray, or a coarray pointer.
- A **coarray pointer** is a pointer to a coarray called a **target coarray**. An ultimate component of a non-coarray structure can also be a coarray pointer.

A static coarray is allocated previously and is static during the program execution. The coshape of a static coarray is explicitly specified in the declaration of the variable. An allocatable coarray is dynamically allocated and deallocated at the ALLOCATE and DEALLOCATE statements. A coarray pointer is dynamically allocated and freed by the intrinsic functions. The coshapes of them are determined at the allocation time and are retained until the deallocation/freeing time without regard to the scoping units. A dummy coarray is an allocated object. The coshape of a dummy coarray is explicitly re-specified in the declaration of the variable. Even if the corresponding actual argument is an allocatable coarrray or a coarray pointer, the specification of the coshape is valid during the execution in the scope.

Coarray Container

A non-coarray structure object can have ultimate (leaf) components as allocatable coarrays (in XcalableMP Fortran) or as coarray pointers (in XcalableMP C). The strucure object is called a **coarray container** in XcalableMP.

[F] A coarray container must be a scalar, may not be a pointer or an allocatable, may not be a coarray, and may not be a function result.

[C] A coarray container may not be a coarray and may not be a function result.

Cosubscript

A coarray on the different image can be referenced and defined by referring the coarray with **cosubscripts**, which is an array element of an image array. See Section **??** for the detail.

[1] A component is an ultimate component of the structure if it is of a basic type or is allocatable or a pointer. An ultimate component of a component is an ultimate component of the structure, recursively.

G.2 Declaration of Coarrays

G.2.1 Declaration Statement of Coarray

Synopsis

Declarations of a variable are extended to add the codimension attribute.

[F] The *type-declaration-stmt* is extended with *coarray-spec*. An *object-name* declared with a *coarray-spec* is a coarray.

[C] the *init-declarator* of the *declaration* and the *parameter-declaration* are extended with a *coarray-spec*. If the *declarator* of them does not start with `*`, the *identifier* declared with a *coarray-spec* is a coarray. Else, the *identifier* declared with a *coarray-spec* is a coarray pointer that may point an unnamed coarray.

Syntax [F]

coarray-spec is adopted in the type declaration statement, as shown below. As a kind of *attr-spec*, the CODIMENSION attribute specifier is added. Besides, *entry-decl* is extended with *coarray-spec*. Underlined parts are addition to the Fortran 90 standard.

type-declaration-stmt **is** *declaration-type-spec [[, attr-spec]... ::] entity-decl-list*

attr-spec **is** ...
...
or CODIMENSION [*coarray-spec*]

entity-decl **is** *object-name [(array-spec)] [[coarray-spec]]* ▮
▮ *[* char-length] [initialization]*
or *function-name [* char-length]*

coarray-spec is defined as follows:

coarray-spec **is** *explicit-coshape-spec*
or *deferred-coshape-spec*

explicit-coshape-spec and *deferred-coshape-spec* is defined in Section G.2.2 and Section G.2.3, respectively.

Syntax [C]

coarray-spec is adopted in *declaration*, as an underlined part shown below:

declaration **is** *declaration-specifiers [init-declarator-list]* ;
init-declarator **is** *declarator [: coarray-spec] [= initializer]*

parameter-declaration **is** *declaration-specifiers declarator [: coarray-spec]*
or *declaration-specifiers [abstract-declarator]*

declarator **is** *[pointer] direct-declarator*

pointer **is** * *[type-qualifier]...*
or * *[type-qualifier]... pointer*

direct-declarator **is** *identifier*
or `(` *declarator* `)`
or *direct-declarator* `[` *[type-qualifier]...* *[assignment-expression]* `]`
or *direct-declarator* `[ static` *[type-qualifier]...* *assignment-expression* `]`
or *direct-declarator* `[` *type-qualifier...* `static` *assignment-expression* `]`
or *direct-declarator* `[` *[type-qualifier]...* `* ]`
or *direct-declarator* `(` *parameter-type-list* `)`
or *direct-declarator* `(` *[identifier-list]* `)`

coarray-spec is defined as follows:

coarray-spec **is** *explicit-coshape-spec*
or *deferred-coshape-spec*

Constraints `[F]`

1. A coarray shall be a dummy argument or have the ALLOCATABLE or SAVE attribute.[2]
2. A coarray shall not be a function result.
3. A coarray shall not be a named constant or a pointer.
4. A coarray shall not be a *common-block-object* or a *equivalence-object*.
5. The VOLATILE attribute shall not be specified for a coarray that is associated by use or host association.
6. Within a BLOCK construct, the VOLATILE attribute shall not be specified for a coarray that is not a construct entity of that construct.

Constraints `[C]`

1. A coarray shall be a dummy argument or have the `static` or `extern` storage class.[3]
2. A coarray shall not be a function. A coarray pointer shall not be a pointer to a function.
3. A coarray shall not be of an `enum` or `union` type.
4. A *declaration-specifiers* of a *declaration* or *parameter-declaration* shall not contain the `volatile` type qualifier.[4]
5. For a coarray pointer, the most right *pointer* of a *declarator* shall not contain the `volatile` type qualifier.[5]
6. For an array coarray, the outermost (the most left) brakets shall not contain the `volatile` type qualifier.

[2] In other words, a local coarray to the procedure should be a SAVE'd or allocatable variable unless it is a dummy argument.

[3] Conversely, a coarray may not have the `auto` or `register` storage class.

[4] Because it is difficult to allow the access to the coarray from outside of the language system. The `const` qualifier here asserts the coarray data is not be modified like INTENT(IN) of Fortran.

[5] It is difficult to allow coarrays to be disturbed from the outside of the language.

Description

[F] The entity declared with a *coarray-spec* is a coarray, and the *coarray-spec* specifies the coshape of the coarray. A coarray is a scalar or an array data object and is of a basic or derived type. The specification of the *coarray-spec* in the *entity-decl* overrides the specification of the *coarray-spec* in the *attr-spec* if both are specified.

[C] The entity of a basic or structure type declared with a *coarray-spec* is a coarray, and the *coarray-spec* specifies the coshape of the coarray. The entity of a pointer type declared with a *coarray-spec* is a coarray pointer, and the *coarray-spec* specifies the coshape of the coarray that the coarray pointer points.

A coarray can be initialized. Each image can initialize coarrays on the image and cannot initialize any coarrays on the other images.

A declaration of a coarray has either an *explicit-coshape-spec* or a *deferred-coshape-spec* as the *coarray-spec*. A static coarray and a dummy coarray are declarad with an *explicit-coshape-spec* (Section G.2.2). An allocatable coaray and a coarray pointer is declarad with a *deferred-coshape-spec* (Section G.2.3).

G.2.2 Explicit coshape

Synopsis

A static coarray and a dummy coarray (Section G.1) are declared with a *coarray-spec* that is an *explicit-coshape-spec*.

Syntax [F]

explicit-coshape-spec **is** *[[lower-cobound :] upper-cobound,]... [lower-cobound :] *****

lower-cobound **is** *specification-expr*

upper-cobound **is** *specification-expr*

Syntax [C]

explicit-coshape-spec **is** `[*]` *[* `[` *coextent* `]` *]...*

coextent **is** *assignment-expression*

Constraints [F]

1. A nonallocatable coarray shall have a *coarray-spec* that is an *explicit-coshape-spec*.
2. A lower-cobound or upper-cobound that is not a constant expression shall appear only in a subprogram, BLOCK construct, or interface body.
3. The upper cobound shall not be less than the lower cobound.

Constraints [C]

1. If a *coarray-spec* appearing in an *init-declarator* or a *parameter-declaration* is an *explicit-coshape-spec*, the *declarator* followed by the *coarray-spec* shall be in the following format:[6]

   declarator **is** *identifier [* `[` *assignment-expression* `]` *]...*

2. A *coextent* that is not a constant expression shall appear only in a block scope or function prototype scope.

[6] Keyword `static` and *assignment-expression*s in "`[ ]`" are not yet taken into account!

Description

An explicit coshape is valid for the scope it is specified. Non-constant values in the explicit coshape are evaluated at the entry of the block in XcalableMP Fortran or at the executon of the declarattion in XcalableMP C.

`[F]` An explicit coshape specifies the corank and the cobounds, except the upper cobound of the final (outermost) dimension. If the lower cobound is omitted, the default value is 1. For each dimension,
(coextent) = (upper cobound) − (lower cobound) + 1.

`[C]` An explicit coshape specifies the corank and the coextents, except the extent of the final (outermost) dimension. The lower cobound is always zero and the upper cobound is the same as the coextent for each dimension.

Example

`[F]` The type declaration statement shown below declares a two-dimensional array coarray of real type. The *coarray-spec* is the notation "`[4,0:*]`". The corank is 2, the lower and upper cobounds of the first dimension are 1 (default) and 4, and the lower cobound of the second dimension is 0.

`[C]` The declaration shown below declares a two-dimensional array coarray of float type. The *coarray-spec* is the notation "`[*][4]`". The corank is 2 and the coextents of the first dimension is 4. Because the lower cobound is always 0, the lower and upper cobounds of the first dimension is 0 and 3, respectively.

XcalableMP Fortran

```
real, save :: z(10,20)[4,0:*]
```

XcalableMP C

```
static float z[20][10]:[*][4];
```

G.2.3 Deferred coshape

Synopsis

An allocatable coarray and a coarray pointer (Section G.1) are declared with a *coarray-spec* that is an *deferred-coshape-spec*.

Syntax

`[F]` *deferred-coshape-spec* **is** *[*`:,` *]*`...` `:`

`[C]` *deferred-coshape-spec* **is** `[ ] ...`

Constraints `[F]`

1. A coarray with the ALLOCATABLE attribute shall have a *coarray-spec* that is a *deferred-coshape-spec*.

Constraints `[C]`

1. If a *coarray-spec* appearing in an *init-declarator* or a *parameter-declaration* is an *deferred-coshape-spec*, the *declarator* followed by the *coarray-spec* shall be a pointer.

Description

A deferred coshape specifies the corank of the coarray but does not specify the cobounds or the coextents, which are determined by allocation or argument association. If the coarray or the coarray pointer is a dummy argument, the coshape is inherited from the corresponding actual argument.

Example

[F] The type declaration statement shown below declares a two-dimensional array coarray of real type. The *coarray-spec* of `z` is the notation "`[:,:]`", and the corank is 2. The *coarray-spec* of `x` is the notation "`[:]`", and the corank is 1. All upper and lower cobounds are always unknown in deferred coshape.

[C] The declaration shown below declares a two-dimensional array coarray of float type. The *coarray-spec* of `z` is the notation "`[][]`", and the corank is 2. The *coarray-spec* of `x` is the notation "`[]`", and the corank is 1. All coextents are always unknown in deferred coshape.

XcalableMP Fortran

```
real, allocatable :: z(:,:)[:,:]
type(my_t), allocatable :: x[:]
```

XcalableMP C

```
float (*z)[10]:[][];
struct my_t *x:[];
```

G.2.4 Coarray Container

[F] A derived type that has a scalar coarray component is called a **coarray container type**. A derived type that has a scalar component of a coarray container type is also called a coarray container type. A scalar object of a coarray container type is called a **coarray container**.

[C] A `struct` type that has a coarray component is called a **coarray container type**. A `struct` type that has a component of a coarray container type is also called a coarray container type. A object of a coarray container type is called a **coarray container**.

Constraints [F]

1. A coarray container shall be a dummy argument or have the ALLOCATABLE or SAVE attribute.[7]

2. A coarray container shall be a nonpointer nonallocatable scalar, shall not be a coarray, and shall not be a function result.

Constraints [C]

1. A coarray container shall be a dummy argument or have the `static` or `extern` storage class.[8]

2. A coarray container shall not be a coarray.

Example

[F] Variable `cc` is a structure container and has a coarray structure component `cc%a`. Variable `dd` is also a structure container and has ultimate coarray structure components `dd%x%a` and `dd%y%a`.

[7] In other words, a local coarray container to the procedure should be a SAVE'd or allocatable structure unless it is a dummy argument.

[8] Conversely, a coarray container may not have `auto` storage class.

[C] Variable cc is a structure container and has a coarray structure component cc.a. Variable dd is also a structure container and has ultimate coarray structure components dd.x.a and dd.y.a.

XcalableMP Fortran

```
type cc_t
  integer size
  real, allocate :: a(:)[:,:]
end type
type dd_t
  type(cc_t) :: x,y
end type
type(cc_t),save :: cc
type(dd_t),save :: dd
```

XcalableMP C

```
typedef struct {
  int size;
  double (*a)[10]:[][];
} cc_t;
typedef struct {
  cc_t x, y;
} dd_t;
static cc_t cc;
static dd_t dd;
```

G.3 Argument Association

Constraints [F]

1. An entity with the INTENT(OUT) attribute shall not be an allocatable coarray or a coarray container.

2. An entity with the VALUE attribute shall not be a coarray container.

3. A procedure that has a coarray dummy argument shall have an explicit interface if it is referenced.

Example

[F] a1 and a2 are explicit-shape coarrays and a3 and a4 are assumed-shape coarrays. These coarrays must have explicit coshapes. Because subroutine foo has coarray dummy arguments, the explicit interface must be visible to subroutine caller.

[C] a1 and a2 are specified size of coarrays and a3 and a5 are unspecified size of coarrays. These coarrays must have explicit coshapes. Because function foo has coarray dummy arguments, the prototype definition must be visible to function caller.

XcalableMP Fortran

```
module moo
 integer,parameter :: m=5
end module moo

!-- CALLER --
subroutine caller
 interface
  subroutine foo(n, a1, a2, a3, a4)
   use moo
   integer n
   real a1(10,5)[*], a2(n,m)[*]
   real a3(10,*)[*], a4(0:n-1,0:*)[*]
  end subroutine foo
 end interface
 real,save :: a(10,5)[*]
 call foo(10, a, a, a, a)
end subroutine caller

!-- SUBROUTINE --
subroutine foo(n, a1, a2, a3, a4)
 use moo
 integer n
 real a1(10,5)[*], a2(n,m)[*]
 real a3(10,*)[*], a4(0:n-1,0:*)[*]
 ...
end subroutine caller
```

XcalableMP C

```
int const m = 5;

/*-- PROTOTYPE --*/
void foo(int n,
         float a1[5][10]:[*],
         float a2[m][n]:[*],
         float a3[][10]:[*],
         float a5[][*]:[*]);

/*-- CALLER --*/
void caller() {
  float static a[5][10];

  foo(10, a, a, a, a);
}

/*-- FUNCTION --*/
void foo(int n,
         float a1[5][10]:[*],
         float a2[m][n]:[*],
         float a3[][10]:[*],
         float a5[][n]:[*]) {
  ...

}
```

G.4 Memory Allocation of Coarrays

G.4.1 [F] Allocation of allocatable coarray

TBD

G.4.2 [C] Allocation of coarray pointer

A coarray pointer is a pointer to a coarray object that is called a target coarray. A target coarray is allocated and freed with library functions `xmp_comalloc` and `xmp_cofree`, respectively. A coarray pointer retains the address of an allocated target coarray. To avoid aliasing between coarrays, a coarray pointer is not allowed to point to any named coarrays or its subobjects, nor to point the same target coarray or its subobjects that is pointed from the other coarray pointer.

Example

The first line of the code fragment shown below declares coarray pointer **y** pointing to an array coarray of double type. The second line allocates a target coarray of size `sizeof(double)*10*20` with the first-dimension coextent 4. The third line frees the coarray and the value of **y** becomes invalid.

XcalableMP C

```
double (*y)[10]:[][];
y = xmp_comalloc(sizeof(double)*10*20, 4);
xmp_cofree(y);
```

G.5 Referrence and Definition to Remote Coarrays

TBD

G.6 Synchronization and Error Handling

TBD

G.7 Intrinsic Procedures

TBD

G.8 Compatibility with the Fortran Standard

TBD

Index