XcalableMP
\textit{\langle ex-scalable-em-p \rangle}
Language Specification

Version 1.2.1

XcalableMP Specification Working Group

November, 2014
History


- 3.3.4 align Directive
- 3.4.3 loop Directive
- 4.7.1 [C] Declarations of Coarrays
- 4.8.1 [F] local_alias Directive
- 5.1 Array Section Notation
- 5.2 Array Assignment Statement

Version 1.2: November 20, 2013

Version 1.1: November 13, 2012

Version 1.0: November 14, 2011
Contents

1 Introduction ............................................. 1
  1.1 Features of XcalableMP ......................................... 1
  1.2 Scope ...................................................... 2
  1.3 Organization of this Document ................................ 2
  1.4 Changes from Version 1.2 .................................... 2

2 Overview of the XcalableMP Model and Language .............. 3
  2.1 Hardware Model ............................................ 3
  2.2 Execution Model ........................................... 3
  2.3 Data Model ............................................... 4
  2.4 Global-view Programming Model ............................. 4
  2.5 Local-view Programming Model ............................. 5
  2.6 Interactions between the Global View and the Local View . 6
  2.7 Base Languages ........................................... 6
  2.8 Glossary .................................................... 7
     2.8.1 Language Terminology ................................ 7
     2.8.2 Node Terminology ..................................... 8
     2.8.3 Data Terminology ...................................... 10
     2.8.4 Work Terminology ..................................... 10
     2.8.5 Communication and Synchronization Terminology .... 10
     2.8.6 Local-view Terminology ............................... 11

3 Directives ............................................... 13
  3.1 Directive Format .......................................... 13
     3.1.1 General Rule ......................................... 13
     3.1.2 Combined Directive .................................... 15
  3.2 nodes Directive ........................................... 15
     3.2.1 Node Reference ....................................... 17
  3.3 Template and Data Mapping Directives .................... 18
     3.3.1 template Directive .................................... 18
     3.3.2 Template Reference ................................... 19
     3.3.3 distribute Directive ................................ 20
     3.3.4 align Directive ....................................... 22
     3.3.5 shadow Directive ...................................... 25
     3.3.6 template.fix Construct ............................... 26
  3.4 Work Mapping Construct .................................. 27
     3.4.1 task Construct ......................................... 27
     3.4.2 tasks Construct ........................................ 29
     3.4.3 loop Construct ......................................... 31
     3.4.4 array Construct ....................................... 38
3.5 Global-view Communication and Synchronization Constructs

3.5.1 reflect Construct
3.5.2 gmove Construct
3.5.3 barrier Construct
3.5.4 reduction Construct
3.5.5 bcast Construct
3.5.6 wait_async Construct
3.5.7 async Clause

4 Support for the Local-view Programming

4.1 Rules Determining Image Index

4.1.1 Primary Image Index
4.1.2 Image Index Determined by a task Directive
4.1.3 Current Image Index
4.1.4 Image Index Determined by a Non-primary Node Array
4.1.5 Image Index Determined by an Equivalenced Node Array
4.1.6 On-node Image Index

4.2 Basic Concepts

4.3 coarray Directive

4.3.1 Purpose and Form of the coarray Directive
4.3.2 An Example of the coarray Directive

4.4 image Directive

4.4.1 Purpose and Form of the image Directive
4.4.2 An Example of the image Directive

4.5 Image Index Translation Intrinsic Procedures

4.5.1 Translation to the Primary Image Index
4.5.2 Translation to the Current Image Index

4.6 Examples of Communication between Tasks

4.7 Coarrays in XcalableMP C

4.7.1 [C] Declaration of Coarrays
4.7.2 [C] Reference of Coarrays
4.7.3 [C] Synchronization of Coarrays

4.8 Directives for the Local-view Programming

4.8.1 [F] local alias Directive
4.8.2 post Construct
4.8.3 wait Construct
4.8.4 [C] lock/unlock Construct

5 Base Language Extensions in XcalableMP C

5.1 Array Section Notation
5.2 Array Assignment Statement
5.3 Built-in Functions for Array Section
5.4 Pointer to Global Data

5.4.1 Name of Global Array
5.4.2 The Address-of Operator

5.5 Dynamic Allocation of Global Data
5.6 The Descriptor-of Operator
7 Intrinsic and Library Procedures

7.1 [F] Intrinsic Functions

7.1.1 xmp_desc_of

7.2 System Inquiry Functions

7.2.1 xmp_all_node_num
7.2.2 xmp_all_num_nodes
7.2.3 xmp_node_num
7.2.4 xmp_num_nodes
7.2.5 xmp_wtime
7.2.6 xmp_wtick

7.3 Synchronization Functions

7.3.1 xmp_test_async

7.4 Memory Allocation Functions

7.4.1 [C] xmp_malloc

7.5 Mapping Inquiry Functions

7.5.1 xmp_nodes_ndims
7.5.2 xmp_nodes_index
7.5.3 xmp_nodes_size
7.5.4 xmp_nodes_attr
7.5.5 xmp_nodes_equiv
7.5.6 xmp_template_fixed
7.5.7 xmp_template_ndims
7.5.8 xmp_template_lbound
7.5.9 xmp_template_ubound
7.5.10 xmp_dist_format
7.5.11 xmp_dist_blocksize
7.5.12 xmp_dist_gblockmap
7.5.13 xmp_dist_nodes
7.5.14 xmp_dist_axis
7.5.15 xmp_align_axis
7.5.16 xmp_align_offset
7.5.17 xmp_align_replicated
7.5.18 xmp_align_template
7.5.19 xmp_array_ndims
7.5.20 xmp_array_lshadow
7.5.21 xmp_array_ushadow
7.5.22 xmp_array_lbound
7.5.23 xmp_array_ubound

7.6 [F] Array Intrinsic Functions of the Base Language

7.7 [C] Built-in Elemental Functions

7.8 Intrinsic/Built-in Transformational Procedures

7.8.1 xmp_scatter
7.8.2 xmp_gather
8 OpenMP in XcalableMP Programs

Bibliography

A Programming Interface for MPI

B Interface to Numerical Libraries

C Memory-layout Model

D XcalableMP I/O

E Sample Programs
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Hardware Model</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>Parallelization by the Global-view Programming Model</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Local-view Programming Model</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>Global View and Local View</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of Shadow of a Two-dimensional Array</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of Periodic Shadow Reflection</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>Sequence Association with a Global Dummy Argument</td>
<td>75</td>
</tr>
<tr>
<td>6.2</td>
<td>Sequence Association with a Local Dummy Argument</td>
<td>76</td>
</tr>
<tr>
<td>6.3</td>
<td>Sequence Association of a Section of a Global Data as an Actual Argument with a Local Dummy Argument</td>
<td>77</td>
</tr>
<tr>
<td>6.4</td>
<td>Sequence Association of an Element of a Global Data as an Actual Argument with a Local Dummy Argument</td>
<td>78</td>
</tr>
<tr>
<td>6.5</td>
<td>Sequence Association with a Global Dummy Argument that Has Full Shadow</td>
<td>78</td>
</tr>
<tr>
<td>6.6</td>
<td>Descriptor Association with a Global Dummy Argument</td>
<td>80</td>
</tr>
<tr>
<td>6.7</td>
<td>Descriptor Association with a Local Dummy Argument</td>
<td>81</td>
</tr>
<tr>
<td>6.8</td>
<td>Passing to a Global Dummy Argument</td>
<td>82</td>
</tr>
<tr>
<td>6.9</td>
<td>Passing to a Local Dummy Argument</td>
<td>83</td>
</tr>
<tr>
<td>6.10</td>
<td>Passing an Element of a Global Data as an Actual Argument to a Local Dummy Argument</td>
<td>83</td>
</tr>
<tr>
<td>B.1</td>
<td>Invocation of a Library Routine through an Interface Procedure</td>
<td>111</td>
</tr>
<tr>
<td>C.1</td>
<td>Example of Memory Layout in the Omni XcalableMP compiler</td>
<td>118</td>
</tr>
</tbody>
</table>
List of Tables

7.1 Built-in Elemental Functions in XcalableMP C .............................................. 100
D.1 Global I/O ........................................................................................................ 120
D.2 Operations for I/O ........................................................................................... 123
Acknowledgment

The specification of XcalableMP 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
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 specification of XcalableMP, 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 “global-view” programming model, and enables parallelizing the original sequential code using minimal modification with simple directives, like 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 of communication and synchronization are taken by directives (and coarray features), which is different from automatic parallelizing compilers. The user should be aware of what happens by the XcalableMP directives in the execution model on the distributed memory architecture.

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

- Extension of existing base languages with directives is useful to reduce code-rewriting and education costs. The XcalableMP language specification is defined on Fortran or C as a base language.

- For flexibility and extensibility, the execution model allows to combine 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 obtained in the development of HPF, HPF/JA [1], Fujitsu XPF (VPP FORTRAN) [2, 3], and OpenMPD [7].
1.2 Scope

The XcalableMP specification covers only user-directed parallelization, wherein 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 lay out data, detect parallelism and 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: Directives
- Chapter 4: Support for the Local-view Programming
- Chapter 5: Base Language Extensions in XcalableMP C
- 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 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.
- 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 each other via an interconnection network. Each node can access its local memory directly and remote memory, that is, the memory of another node indirectly (i.e. via communication). However, it is assumed that accessing remote memory is much slower than accessing local memory.

![Figure 2.1: Hardware Model](image)

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 keeps executing the same code independently (i.e. asynchronously), which is referred to as the *replicated execution*, until it encounters an XcalableMP construct.

3
A set of nodes that executes a procedure, a statement, a loop, a block, etc. is referred to as its \textit{executing node set} and determined by the innermost \texttt{task}, \texttt{loop} or \texttt{array} directive surrounding it dynamically, or at runtime. The \textit{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 \textit{primary node set}, is a node set that contains all the available nodes, which can be specified in an implementation-dependent way (e.g. through a command-line option).

When a node encounters at runtime either a \texttt{loop}, \texttt{array}, or \texttt{task} construct, and is contained by the node set specified by the \texttt{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 following statements.

Particularly when a node in the current executing node set encounters a \texttt{loop} or an \texttt{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 where 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 \textit{global constructs} are performed collectively by the current executing nodes. Note that neither synchronizations nor communications occur without these constructs specified.

## 2.3 Data Model

There are two classes of data in XcalableMP: \textit{global data} and \textit{local data}. Data declared in an XcalableMP program are local by default.

Global data are ones that are distributed onto the executing node set by the \texttt{align} directive (see section \ref{sec:3.3.4}). Each fragment of a global data is allocated in the local memory of a node in the executing node set.

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

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

Particularly in XcalableMP Fortran, for common blocks that include any global variables, the ways how the storage sequence of them is defined and how the storage association of them is resolved are implementation-dependent.

## 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 the data among nodes using the data distribution directives. The \texttt{loop} construct assigns each iteration of a loop to the node where the computed data is located. The global-view communication directives are used to synchronize nodes, to maintain the consistency of the shadow area, and to move part of the distributed data globally. Note that the programmer must specify explicitly communications to make all data reference in the program local by using appropriate directives.

In many cases, the XcalableMP program according to the global-view programming model is based on a sequential program and can produce the same results as it, regardless of the number
of nodes (Figure 2.2).

There are three groups of directives for the global-view programming model. Since 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 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 specified data elements. The _task_ construct defines an amount of work as a _task_ and assigns it to a specified 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.

---

**2.5 Local-view Programming Model**

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

For the local-view programming model, some language extensions and directives are provided. The coarray notation imported from Fortran 2008 is one of such extensions 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 communication to obtain the value from remote memory (i.e. *get* operation) occurs. If the access is a definition, then communication to set a value to remote memory (i.e. *put* operation) occurs.

![Figure 2.3: Local-view Programming Model](image)

2.6 Interactions between the Global View and the Local View

In the global view, nodes are used to distribute data and computational load. In the local view, nodes are used to address data in the coarray notation. In the application program, 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 so that they can be accessed also 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 on Fortran or C as a base language. 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.
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.
  - **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, but instead 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, barrier, etc., 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 abstract, for example, a set of grid points in the grid method or 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 is 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 model of programming or parallelization, on which parallel programs are written by specifying how to map global data and works onto nodes.

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

primary node set An entire node set that is specified explicitly or implicitly, and is the current executing node set 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, a statement, a construct, etc. of an XcalableMP program is called its executing node set. This term is used in this document to represent the current executing node set unless it is ambiguous. Note that the executing node set of the main routine is the primary 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.

non-primary node array A node array declared without “=node-ref”, “=*”, or “=” in a NODES directive. A non-primary node array corresponds to all the nodes at the invocation of a program, and also corresponds to all the images at the invocation of a program.

primary node array A node array declared with the rhs of a node reference by “*” representing the primary node set. A primary node array corresponds to all the nodes at the invocation of a program, and also corresponds to all the images at the invocation of a program.

executing node array A node array declared with the rhs of a node reference by “*” representing the executing node set. An executing node array corresponds to the executing node set, and also corresponds to the current set of images at the evaluation of the declaration of the node array.

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

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

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

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

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

**local section** A section of a global data that is allocated as an array on each node at runtime. The local section of a global data 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** A data movement among nodes. Communication in XcalableMP occurs only when the programmer instruct it explicitly with a global communication construct or a coarray reference.

**reduction** A procedure of 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 among 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 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, that is, a distributed array. A local alias can be used in XcalableMP programs in the same way as normal local data.

current set of images The current set of images is a set of images determined by the most lately executed task-directive in the TASK directive constructs that are not completed if any TASK directive constructs are being executed. The current set of images is all the images at the invocation of a program if there are no TASK directive constructs that are not completed.

image An instance of an XcalableMP program. Each image uniquely corresponds to a node.

image index An integer value identifying an image in a set of images.

In XcalableMP C, the lower cobound in each axis is one by default and taking account of the cobound, the cosubscript list in an image selector determines the image index in the same way that a subscript list in an array element determines the subscript order value in Fortran, taking account of the bounds.
Chapter 3

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.
[F] The syntax rule continues.
[C] The following lines are effective only in XcalableMP Fortran.
[C] The following lines are effective only in XcalableMP C.

3.1 Directive Format

3.1.1 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 overridden in the inner blocks or use or host associated, and, in XcalableMP C, a node array declared by a nodes directive is visible only in the range from the declaring point to

\(^1\)Consequently, the rules of comment lines that an XcalableMP directive follows is the same as the ones that an OpenMP directive follows.
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 unit 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 compound.

The following directives are executable directives.

- template_fix
- task
- tasks
- loop
- array
- reflect
- gmove
- barrier
- reduction
- bcast
- wait_async
3.1.2 Combined Directive

Synopsis

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

Syntax

\[
[F] \!$xmp \text{combined-directive} \text{ is } \text{combined-attribute [, combined-attribute ]... :: combined-decl [, combined-decl ]...}
\]

\textit{combined-attribute} is one of:

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

and \textit{combined-decl} is one of:

- \texttt{nodes-decl}
- \texttt{template-decl}
- \texttt{array-name}

Description

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

Restrictions

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

3.2 nodes Directive

Synopsis

The \texttt{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 ]... )

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 form of the nodes directive is used to declare a node array that corresponds to the entire node set. The second form is used to declare a node array, each node of which is assigned to a node of the node set specified by nodes-ref at the corresponding position in Fortran’s array element order, as if the node set were a one-dimensional node array.

If node-size in the last dimension is “*”, then the size of the node array is automatically adjusted according to the total size of the entire node set in the first form, the executing node set in the second form, or the referenced node set in the third form.

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.
- 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 form, the executing node set in the second form, or the referenced node set in the third form.
- nodes-subscript in nodes-ref must not be “*”.

Examples

The following are examples of the first and the third forms appeared in the main program. Since 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.
The following is an example of a node declaration in a procedure. Since p is declared in the second form to be of size 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 form 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.

```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
```

3.2.1 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, the "*" symbol or "**".

```
    nodes-ref is nodes-name [(nodes-subscript [, nodes-subscript] ... )]
    or *
    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, which is totally ordered in Fortran’s array element order. A node reference by "*" represents the executing node set. A node reference by "**" represents the primary node set.

Specifically, the "*" symbol appeared 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 \texttt{distribute} directive,
  \begin{verbatim}
  !$xmp distribute a(block) onto p
  \end{verbatim}

• To specify a node set to which the declared node array corresponds in the second form of the \texttt{nodes} directive,
  \begin{verbatim}
  !$xmp nodes r(2,2,4) = p(1:4,1:4)
  !$xmp nodes r(2,2,4) = (1:16)
  \end{verbatim}

• To specify a node array that corresponds to the executing node set of a task in the \texttt{task} directive,
  \begin{verbatim}
  !$xmp task on p(1:4,1:4)
  !$xmp task on (1:16)
  !$xmp task on p(:,*)
  !$xmp task on (m)
  \end{verbatim}

• To specify a node array with which iterations of a loop are aligned in the \texttt{loop} directive,
  \begin{verbatim}
  !$xmp loop (i) on p(lb(i):lb(i+1)-1)
  \end{verbatim}

• To specify a node array that corresponds to the executing node set in the \texttt{barrier} and the \texttt{reduction} directive,
  \begin{verbatim}
  !$xmp barrier on p(5:8)
  !$xmp reduction (+:a) on p(*,*)
  \end{verbatim}

• To specify the source node and the node array that corresponds to the executing node set in the \texttt{bcast} directive,
  \begin{verbatim}
  !$xmp bcast (b) from p(k) on p(:)
  \end{verbatim}

\section{Template and Data Mapping Directives}

\subsection{template Directive}

\textbf{Synopsis}

The \texttt{template} directive declares a template.

\textbf{Syntax}

\begin{verbatim}
[F]  !$xmp template template-decl [, template-decl ]...

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

where template-decl is:

\begin{verbatim}
  template-name ( template-spec [, template-spec ]... )
\end{verbatim}

and template-spec must be one of:

\begin{verbatim}
  [int-expr :] int-expr
\end{verbatim}
3.3. TEMPLATE AND DATA MAPPING DIRECTIVES

Description

The template directive declares a template with the shape specified by the sequence of template-spec's. If every template-spec 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 3.3.6) at runtime. If int-expr is specified as template-spec, then 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 “:”.

3.3.2 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

\[
\text{template-ref} \text{ is } \text{template-name} \left( \text{template-subscript} \left[ \text{template-subscript} \right] \ldots \right)
\]

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 where the specified subset of the template resides.

Specifically, the “*” symbol appeared 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 with a template reference in the on clause.
  \[
  \text{!$xmp task on t(1:m,1:n)}
  \]
  \[
  \text{!$xmp task on t}
  \]

- In the loop directive, the executing node set of each iteration of the following loop is indirectly specified with a template reference in the on clause.
  \[
  \text{!$xmp loop (i) on t(i-1)}
  \]
In the array directive, the executing node set on which the following array assignment statement is performed in parallel is indirectly specified with a template reference in the on clause.

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

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

```plaintext
!$xmp barrier on t(1:n)
!$xmp reduction (+:a) on t(*,:)
!$xmp bcast b from p(k) on t(1:n)
```

### 3.3.3 distribute Directive

**Synopsis**

The distribute directive specifies distribution of a template.

**Syntax**

```plaintext
[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
```

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 appearing in the onto clause corresponds, in left-to-right order, with 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)`.
3.3. TEMPLATE AND DATA MAPPING DIRECTIVES

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 non-negative 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 its reference in the executable code in the block.

Examples

Example 1

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

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

<table>
<thead>
<tr>
<th>p(1)</th>
<th>t(1:16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(2)</td>
<td>t(17:32)</td>
</tr>
<tr>
<td>p(3)</td>
<td>t(33:48)</td>
</tr>
<tr>
<td>p(4)</td>
<td>t(49:64)</td>
</tr>
</tbody>
</table>
Example 2

```fortran
!$xmp nodes p(4)
!$xmp template t(64)
!$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>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p(1)</td>
<td>t(1:8)</td>
<td>t(33:40)</td>
</tr>
<tr>
<td>p(2)</td>
<td>t(9,16)</td>
<td>t(41:48)</td>
</tr>
<tr>
<td>p(3)</td>
<td>t(17,24)</td>
<td>t(49:56)</td>
</tr>
<tr>
<td>p(4)</td>
<td>t(25,32)</td>
<td>t(57:64)</td>
</tr>
</tbody>
</table>

Example 3

```fortran
!$xmp nodes p(8,5)
!$xmp template t(64,64,64)
!$xmp distribute t(*,cyclic,block) 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>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p(1,1)</td>
<td>t(1:64, 1:57:8, 1:13)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p(2,1)</td>
<td>t(1:64, 2:58:8, 1:13)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p(8,5)</td>
<td>t(1:64, 8:64:8, 53:64)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the size of the third dimension of `t`, 64, is not divisible by the size of the second dimension of `p`, 5. Thus, sizes of the blocks in the third dimension are different among nodes.

3.3.4 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]... )
```

where `align-source` must be one of:
3.3. TEMPLATE AND DATA MAPPING DIRECTIVES

\[
\text{scalar-int-variable} \\
* \\
: \\
\]

and \textit{align-subscript} must be one of:

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

Note that the variable \textit{scalar-int-variable} appearing in \textit{align-source} is referred to as an “align dummy variable” and \textit{int-expr} appearing in \textit{align-subscript} as an “align offset.”

\textbf{Description}

The array specified by \textit{array-name} is aligned with the template specified by \textit{template-name} so that each element of the array indexed by the sequence of \textit{align-source}’s is aligned with the element of the template indexed by the sequence of \textit{align-subscript}’s, where \textit{align-source}’s and \textit{align-subscript}’s are interpreted as follows:

1. The first form of \textit{align-source} and \textit{align-subscript} represents an align dummy variable and an expression of it, respectively. The align dummy variable ranges over all valid index values in the corresponding dimension of the array.

2. The second form “*” of \textit{align-source} and \textit{align-subscript} represents a dummy variable (not an align dummy variable) that does not appear anywhere in the directive.
   - The second form of \textit{align-source} is said to “collapse” the corresponding dimension of the array. As a result, the index along the corresponding dimension makes no difference in determining the alignment.
   - The second form of \textit{align-subscript} is said to “replicate” the array. Each element of the array is replicated, and aligned to all index values in the corresponding dimension of the template.

3. The third form of \textit{align-source} and the matching \textit{align-subscript} represents a same align dummy variable that ranges over all valid index values in the corresponding dimension of the array. The matching of colons (“:”) in the sequence of \textit{align-source}’s and \textit{align-subscript}’s is determined as follows:
   - \textbf{[F]} Colons in the sequence of \textit{align-source}’s and those in the sequence of \textit{align-subscript}’s are matched up in corresponding left-to-right order, where any \textit{align-source} and \textit{align-subscript} that is not a colon is ignored.
   - \textbf{[C]} Colons in the sequence of \textit{align-source}’s in right-to-left order and those in the sequence of \textit{align-subscript}’s in left-to-right order are matched up, where any \textit{align-source} and \textit{align-subscript} that is not a colon is ignored.

In XcalableMP C, an \textbf{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).

\textbf{Restrictions}

- \textbf{[C]} \textit{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-subscript's.

• 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's must contain exactly as many colons as the sequence of align-subscript's contains.

• [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 its appearance 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)
```

The array element $a(i)$ is aligned with the template element $t(i)$. This is equivalent to the following code.

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

Example 2

```
XcalableMP Fortran
!
$xmp align a(*,j) with t(j)
```

The subarray $a(:,j)$ is aligned with the template element $t(j)$. Note that the first dimension of $a$ is collapsed.

Example 3

```
XcalableMP Fortran
!
$xmp align a(j) with t(*,j)
```

The array element $a(j)$ is replicated and aligned with each template element of $t(:,j)$.

Example 4

```
XcalableMP Fortran
!
$xmp template t(n1,n2)
  real a(m1,m2)
!
$xmp align a(*,j) with t(*,j)
```
The subarray $a(:,j)$ is aligned with each template element of $t(:,j)$.

By replacing “*” in the first dimension of the array $a$ and “*” in the first dimension of the template $t$ with a dummy variable $i$ and $k$, respectively, this alignment can be interpreted as the following mapping.

$$a(i,j) \rightarrow t(k,j) \mid (i,j,k) \in (1:n1, 1:n2, 1:m1)$$

### 3.3.5 shadow Directive

#### Synopsis

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

#### Syntax

[F] \texttt{!$xmp shadow array-name (shadow-width [, shadow-width]...) }

[C] \texttt{#pragma xmp shadow array-name [shadow-width][shadow-width]...}

where $shadow-width$ must be one of:

- \texttt{int-expr}
- \texttt{int-expr : int-expr}
- \texttt{*}

#### 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 “\texttt{int-expr : int-expr},” the shadow area of the width specified by the first \texttt{int-expr} is added at the lower bound and that specified by the second one at the upper bound in the dimension. When \texttt{shadow-width} is of the form \texttt{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 all of the area that it does not own is regarded as shadow. This type of shadow is sometimes referred to as a “full shadow.”

Note that the shadow area of a multi-dimensional array include “obliquely-neighboring” elements, which are the ones owned by the node whose indices are different in more than one dimension, and that the shadow area can be allocated also at the global lower and upper bound 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 it. 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 non-negative 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 its appearance in the executable code in the block.

Example

```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 3.1: Example of 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 3.1). Among them, a(16,16), a(33,16), a(16,33) and a(33,33) are “obliquely-neighboring” elements of p(2,2).

3.3.6 template_fix Construct

Synopsis

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

Syntax

```fortran
[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]... ])
```

where template-spec is:

- [int-expr:] int-expr

and dist-format is one of:

- block ([ int-expr ])
- cyclic ([ int-expr ])
- gblock ( int-array )
3.4. WORK MAPPING CONSTRUCT

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 aligned with an initially undefined template must be an allocatable array, in XcalableMP Fortran, or a pointer (see Section 5.5), in XcalableMP C, which cannot be allocated until the template is fixed by the template_fix construct. Any constructs that have such a template in their on clause must not 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-format's 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-format's exists in a template_fix construct, it must be identical with the sequence of dist-format's in the distribute directive for the template specified by template-name, except for int-array specified in the parenthesis following gblock.

- Either the sequence of dist-format's or the sequence of template-spec's must be given.

Example

```f90
!$xmp template :: t(:)
!$xmp distribute (gblock(*)) :: t
   real , allocatable :: a(:)
!$xmp align (i) with t(i) :: a
   ...
   N = ...; M(...) = ...
   ...
!$xmp template_fix(gblock(M)) t(N)
   ...
   allocate (a(N))
```

Since the shape is (: ) 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.

3.4 Work Mapping Construct

3.4.1 task Construct

Synopsis

The task construct defines a task that is executed by a specified node set.
CHAPTER 3. DIRECTIVES

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 executing 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 entry 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 immediately surrounding tasks construct. The current executing node set is set to that specified by the on clause at the entry of the task construct and 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 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.

```
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(1)
{
   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 $a(\cdot, j)$ or $a[j][]$. 
3.4. WORK MAPPING CONSTRUCT

3.4.2 tasks Construct

Synopsis

The tasks construct is used to instruct the executing nodes to execute the multiple tasks it surrounds in 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 entry of each task. As a result, if there is no overlap 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 entry and exit 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), respectively.
Example 2 The first node \( p(1) \) executes the first and the 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.
3.4.3 loop Construct

Synopsis

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

Syntax

\[ \text{do-loops} \]

\[
\begin{align*}
[F] \!$xmp & \text{ loop } \left( \text{loop-index}, \text{loop-index}\right) \ldots \right) \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \right] \r

where \text{reduction-clause} is:

\[
\text{reduction( reduction-kind : reduction-spec, reduction-spec... )}
\]

\text{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 distribute the execution of their iterations onto the node set specified by the on clause.

The sequence of loop-index’s 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 possible value of it 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-index’s 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-index’s is executed by a node specified by the
3.4. WORK MAPPING CONSTRUCT

In addition, the executing node set is updated to the node set specified by the on clause at the beginning of every iteration and 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.

The reduction operation executed, except in cases with reduction-kind of FIRSTMAX, FIRSTMIN, LASTMAX, or LASTMIN, is equivalent to the reduction construct with the same reduction-kind and 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. Therefore, for example, the two codes below are equivalent.

XcalableMP Fortran

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

s = s op s_tmp
```

Particularly for the reduction kinds of FIRSTMAX, FIRSTMIN, LASTMAX and LASTMIN, in addition to a corresponding MAX or MIN reduction operation, the location-variable's 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 means first 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 means last 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, and each local variable referenced in the directive must have the same value among all of them, and the lower bound, upper bound and step of the associated loop must have the same value among all of them.

• 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

```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 iterations to be executed by each node can be determined before executing the loop.

XcalableMP Fortran

```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

```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
```
Since 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.

Example 3

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

The distribution of loops in the nested loop can be specified using the sequence of 'loop-index's 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.

Example 4

```fortran
!$xmp distribute t(block,block) onto p
!$xmp align (i,j) with t(i,j) :: a, b
...
  !$xmp loop (i,j) on t(i,j)
    do j = 1, M
      do i = 1, N
        a(i,j) = 1.0
        b(i,j) = a(i,j)
      end do
    end do
!$xmp end loop
end do
```
Three node sets \(p(:,1)\), \(p(:,2)\) and \(p(:,3)\) are created as the executing node sets, and each of them executes iteration 1, 2 and 3 of the associated loop, respectively. This example is equivalent to the loop containing \texttt{task} constructs (below left) or static \texttt{tasks/task} constructs (below right).

Example 5

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 \texttt{task} constructs (below left) or static \texttt{tasks/task} constructs (below right).
3.4. WORK MAPPING CONSTRUCT

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
```

This loop computes the maximum value of \( a(i,j) \) and stores it into the variable \( \text{amax} \) 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 with the reduction construct.

### 3.4.4 array Construct

**Synopsis**

The array construct divides the work of an array assignment among 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 parallel execution of an array assignment, where each subassignment and sub-operation of an element is executed by a node determined by the on clause.

Note that array assignments can be used also in XcalableMP C, which is one of the language extensions introduced by XcalableMP (see Section 5.2).

**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 with the associated array assignment.
- The array construct is global and must be executed by all of the executing nodes, and each variable appearing in the construct must have the same value among all of them.

**Examples**

**Example 1**

```
!$xmp distribute t(block) onto p  
!$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.
3.5 Global-view Communication and Synchronization Constructs

3.5.1 reflect Construct

Synopsis

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

Syntax

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

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

where reflect-width must be one of:
The `reflect` construct updates each of the shadow object of the array specified by `array-name` with the value of its corresponding reflection source. Note that the shadow objects corresponding to “obliquely-neighboring” elements can be also updated with this construct.

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

Particularly 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 corresponding to the data object of the global upper (lower) bound and updated with that value 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-name`’s must be mapped onto the executing node set.
- The `reflect` width of each dimension specified by `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**

```
!$xmp nodes p(4)
!$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)
```

The `shadow` directive allocates “periodic” shadow areas of the array `a`. The `reflect` construct updates “periodically” the shadow area of `a` (Figure 3.2). 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)`.
3.5. GLOBAL-VIEW COMMUNICATION AND SYNCHRONIZATION CONSTRUCTS

3.5.2  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 (rhs) variable into the left-hand side (lhs) of the associated assignment statement, which may require 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 cause an implicit synchronization after it among the executing nodes.

  If the async clause is not specified, then the construct is “synchronous” and it is guaranteed that the lhs data can be read and overwritten, the rhs 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 until returning from the associating wait async construct (Section 3.5.6).

- **in mode**
  
  When the in clause is specified, the rhs data of the assignment, whole or parts 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 lhs data can be read and overwritten and all of the operations of the construct on the owner nodes of the rhs and the executing nodes are completed when returning from the construct; otherwise, the construct is “asynchronous” and it is not guaranteed that until returning from the associating wait async construct (Section 3.5.6).

- **out mode**
  
  When the out clause is specified, the lhs data of the assignment, whole or parts 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 rhs data can be overwritten and all of the operations of the construct on the owner nodes of the lhs and the executing nodes are completed when returning from the construct; otherwise, the construct is “asynchronous” and it is not guaranteed that until returning from the associating `wait async` construct (Section 3.5.7).

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 among all of them.
- If the `gmove` construct is in `collective` mode, then all elements of the distributed arrays appearing in both the lhs and the rhs of the associated assignment statement must reside in the executing node set.
- If the `gmove` construct is in `in` mode, then all elements of the distributed array appearing in the lhs of the associated assignment statement must reside in the executing node set.
- If the `gmove` construct is in `out` mode, then all elements of the distributed array appearing in the rhs 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 both the lhs and the rhs are distributed arrays, then the copy operation is performed by all-to-all communication. If the lhs is a replicated array, this copy is performed by multi-cast communication. If the rhs is a replicated array, then no communication is required.

```
XcalableMP Fortran    XcalableMP C
!$xmp gmove             #pragma xmp gmove
  a(:,1:N) = b(:,3,0:N-1)  a[1:N][:] = b[0:N][3][:];
```

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

```
XcalableMP Fortran    XcalableMP C
!$xmp gmove             #pragma xmp gmove
  a(:,1:N) = c(k)          a[1:N][:] = c[k]  
```
Example 3: in mode assignment Since b(3) referenced in the rhs of the gmove construct does not reside in the executing node set (p(1:2)), the construct is executed in in mode. Thus, b(3) is transferred from its owner node p(3) to the executing node set.

It is not guaranteed until p(1:2) returns from the construct that any node can read and overwrite a(1:2) and any relevant operations on p(1:2) and p(3) are completed.

```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
```

3.5.3 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.

3.5.4 reduction Construct

Synopsis

The reduction construct performs a reduction operation among nodes.
CHAPTER 3. DIRECTIVES

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 sets the reduction results to the variables on each of the nodes. Note that some of the reduction operation (FIRSTMAX, FIRSTMIN, LASTMAX, and LASTMIN) that could be specified in the reduction clause of the loop directive cannot be specified in the reduction construct, because their semantics are not defined in 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 implementation-dependent.

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 variable’s must either not be aligned or 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 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.

Examples

Example 1

XcalableMP Fortran

```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 section 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

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

Example 2

XcalableMP Fortran

```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)
```
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.

```fortran
end do
  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. The reduction clause cannot be specified for the loop construct of the outer loop to reduce this overhead, because the node set where the reduction operation specified by a reduction clause of a loop construct is performed is determined from its on clause (see 3.4.3) and the on clause of the outer loop construct is different from that of the inner one. However, this code can be modified with the reduction construct as follows:

```fortran
dimension a(n,n), p(n), w(n)

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

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

### 3.5.5 bcast Construct

#### Synopsis

The bcast construct performs broadcast communication from a specified node.
3.5. GLOBAL-VIEW COMMUNICATION AND SYNCHRONIZATION CONSTRUCTS

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 variable's (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 variable's must either not be aligned or 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.

3.5.6 wait_async Construct

Synopsis

The wait_async construct guarantees 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 blocks and therefore statements following it are not executed until all of the asynchronous communications that are specified by async-id's and issued on the node set specified by the on clause are complete.
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 by 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`.

3.5.7 async Clause

Synopsis

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

Description

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

Example

```
!$xmp reflect (a) async(1)
  S1
!$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 1. 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.
Chapter 4

Support for the Local-view Programming

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

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 might be developed independently, with minimum modifications.

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

- Possibility of the 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 exploit the full potential of a particular system.

4.1 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.

4.1.1 Primary Image Index

Every image has a default image index in all the images at the invocation of a program. In XcalableMP, the default image index is the primary image index and is an integer value in the range one to the number of images at the invocation of a program.
A primary node array corresponds to all the images at the invocation of a program, and also corresponds to all 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.

### 4.1.2 Image Index Determined by a task Directive

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.

### 4.1.3 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 at the evaluation of the declaration of the executing node array. In particular, when there are no task directive constructs that are not completed, the current image index of an image is the same as the primary image index.

### 4.1.4 Image Index Determined by a Non-primary Node Array

A non-primary node array corresponds to all the images at the invocation of a program, and 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 the images at the invocation of a program is the subscript order value of the corresponding element of a non-primary node array 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.

### 4.1.5 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 a 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 in the range one to the cardinality of the set of images.

4.1.6 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. Since 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 the directives affect. In particular, the on-node image index for a primary node array is the primary image index.

4.2 Basic Concepts

In XcalableMP, “all images” in Fortran 2008 changes coupled with the execution of `task` constructs and means 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 lately 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 accessed with the current image index by default. The image index that appears in an executable statement indicates the current image index by default.

4.2.1 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)`.

---

```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,
```

---
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.

```
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,:)*)
end subroutine sub2
```

Note

- The result value of \texttt{xmp\_num\_nodes()} is always the same as that of \texttt{NUM\_IMAGES()}.
- The result value of \texttt{xmp\_node\_num()} is always the same as that of \texttt{THIS\_IMAGE()}.
- In a \texttt{read} statement, an io-unit that is an asterisk identifies an external unit that is preconnected for sequential formatted input only on the image whose primary image index is one.

### 4.3 coarray Directive

#### 4.3.1 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 \texttt{object-name} shall be a name of a coarray declared in the same scoping unit.
- The same \texttt{object-name} shall not appear more than once in coarray directives in a scoping unit.
4.3. COARRAY DIRECTIVE

- 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 a 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 allocated most lately the corresponding argument in the chain of argument associations.

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

- The image index determined by an image selector for an on-node coarray shall be in 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.

4.3.2 An Example of the coarray Directive

```
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).
```

 CHAPTER 4. SUPPORT FOR THE LOCAL-VIEW PROGRAMMING

4.4 image Directive

4.4.1 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 execution of a sync all statement performs a synchronization of all 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.

4.4.2 An Example of the image Directive

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
4.5 Image Index Translation Intrinsic Procedures

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

4.5.1 Translation to the Primary Image Index

\texttt{xmp\_get\_primary\_image\_index(NUMBER,INDEX,PRI\_INDEX,NODE\_DESC)}

\textbf{Description.} Translate image indices to the primary image indices.

\textbf{Class.} Subroutine.

\textbf{Arguments.}

- \texttt{NUMBER} shall be a scalar of type default integer. It is an INTENT(IN) argument.
- \texttt{INDEX} shall be a rank-one array of type default integer. The size of \texttt{INDEX} shall be greater than or equal to the value of \texttt{NUMBER}. It is an INTENT(IN) argument. The value of each element of \texttt{INDEX} shall be in the range one to the size of the node array specified in \texttt{NODE\_DESC} if \texttt{NODE\_DESC} appears. The value of each element of \texttt{INDEX} shall be in the range one to the cardinality of the current set of images if \texttt{NODE\_DESC} does not appear.
- \texttt{PRI\_INDEX} shall be a rank-one array of type default integer. The size of \texttt{PRI\_INDEX} shall be greater than or equal to the value of \texttt{NUMBER}. It is an INTENT(OUT) argument. If \texttt{NODE\_DESC} appears, \texttt{PRI\_INDEX(i)} is assigned the primary image index corresponding to the element of the node array specified in \texttt{NODE\_DESC} whose subscript order value is \texttt{INDEX(i)}; otherwise, \texttt{PRI\_INDEX(i)} is assigned the primary image index corresponding to the image whose current image index is \texttt{INDEX(i)}. 

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 with
: ! 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 with 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
**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/).

```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&
  &5(/4,(/1,2,3,4/),index,xmp_desc_of(subnode))
```

### 4.5.2 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 in 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/).

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

### 4.6 Examples of Communication between Tasks

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

```fortran
module nodes
  !$xmp nodes node(8)=** ! A primary node array
  integer, parameter :: n=2
end module nodes
```
4.6. EXAMPLES OF COMMUNICATION BETWEEN TASKS

| 001 | !$xmp nodes subnodeA(n)=node(1:n)  ! subnodeA is for taskA. |
| 002 | !$xmp nodes subnodeB(8-n)=node(n+1:8) ! subnodeB is for taskB. |
| 003 | endmodule |
| 004 | |
| 005 | module intertask |
| 006 | use nodes |
| 007 | real,save :: dA[*],dB[*] |
| 008 | endmodule |
| 009 | |
| 010 | use nodes |
| 011 | !$xmp tasks |
| 012 | !$xmp task on subnodeA  ! The taskA is invoked on subnodeA. |
| 013 | call taskA |
| 014 | !$xmp end task |
| 015 | !$xmp task on subnodeB  ! The taskB is invoked on subnodeB. |
| 016 | call taskB |
| 017 | !$xmp end task |
| 018 | !$xmp end tasks |
| 019 | end |
| 020 | subroutine taskA |
| 021 | use intertask |
| 022 | :
| 023 | me = this_image()  ! The value of me is i on subnodeA(i). |
| 024 | if(me.eq.1)then |
| 025 | call xmp_get_primary_image_index& ! The value of iyoub & & the value of iyoub & & is n+1. |
| 026 | !$xmp image(node) ! Synchronization between |
| 027 | sync images(iyoub) ! node(1) and node(n+1). |
| 028 | call exchange(dA,dB,iyoub) |
| 029 | !$xmp image(node) ! Synchronization between |
| 030 | sync images(iyoub) ! node(1) and node(n+1). |
| 031 | endif |
| 032 | sync all ! Synchronization within subnodeA. |
| 033 | if(me.ne.1)dA = dA[1] |
| 034 | sync all ! Synchronization within subnodeA. |
| 035 | :
| 036 | end |
| 037 | subroutine taskB |
| 038 | use intertask |
| 039 | :
| 040 | me = this_image()  ! The value of me is i on subnodeB(i). |
| 041 | if(me.eq.1)then |
| 042 | call xmp_get_primary_image_index& ! The value of iyoub & & is 1. |
| 043 | !$xmp image(node) ! Synchronization between |
| 044 | sync images(iyoub) ! node(n+1) and node(1). |
| 045 | call exchange(dB,dA,iyoub) |
| 046 | !$xmp image(node) ! Synchronization between |
| 047 | :
In the following program fragment, two tasks communicate with each other without one-to-one synchronization.

```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

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
```
4.7. [C] Coarrays in XcalableMP C

This section describes the coarray features for XcalableMP C.

4.7.1. [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

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

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

4.7.2 [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.

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

Examples

In the following code, each executing node gets whole of $B$ from the image 10 (that is, the tenth node of the entire node set) and copies it into the local storage for $A$.

```c
int A[10]:[*], B[10]:[*];
A[:] = B[:][10];
```

4.7.3 [C] Synchronization of Coarrays

Synopsis

XcalableMP C provides synchronization functions for coarrays.

Format

```c
void xmp_sync_all(int* status)
void xmp_sync_memory(int* status)
void xmp_sync_image(int image, int* status)
void xmp_sync_images(int num, int* image_set, int* status)
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 memory` 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 success, the `status` is defined with `XMP_STAT_SUCCESS`. A condition where the `status` is defined with `XMP_STAT_STOPPED_IMAGE` is the same as that where the `status` is defined with `STAT_STOPPED_IMAGE` in Fortran 2008. These symbolic constants are defined in "xmp.h". If any other error condition occurs during execution of these functions, the `status` is defined with a value which 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 where target images set is defined.

4.8 Directives for the Local-view Programming

4.8.1 [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 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. Every specification and restriction on coarrays is also applied to an associating local
array that is a coarray except that an associating local array is always declared with \textit{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 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 not
either an allocatable array or an array pointer is declared if it is specified as a \textit{local-array-name}
in the \texttt{local.alias} directive.

In XcalableMP C, the \texttt{address-of} operator applied to global data substitutes for the
\texttt{local.alias} directive (see \texttt{5.2}).

\section*{Restrictions}

- A \texttt{global-array-name} shall be a name of an explicitly mapped array declared in the same
  scoping unit.

- A \texttt{local-array-name} shall be a name of a non-mapped array declared in the same scoping
  unit.

- A \texttt{local-array-name} shall not be a dummy argument.

- An associating local array shall have the declared type and type parameters of the corre-
  sponding associated global array.

- An associating local array shall be declared with \textit{assumed-shape-spec-list} of the same rank
  as the corresponding associated global array.

- A \texttt{local-array-name} shall appear in a \texttt{COARRAY} directive in the same scoping unit and
  the \texttt{node-name} in the \texttt{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
  \texttt{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 \texttt{LOCAL_ALIAS} directive, the dummy argument shall appear in a \texttt{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.

\section*{Examples}

\subsection*{Example 1}

\begin{verbatim}
!$xmp nodes n(4)
!$xmp template :: t(100)
!$xmp distribute (block) onto n :: t
\end{verbatim}
The array \( a \) is distributed by block onto four nodes. The node \( n(2) \) has its local section of twenty-five elements \((a(25:50))\) with shadow areas of size one on both of 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 \( n(2) \).

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower shadow</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>50</td>
<td>26</td>
</tr>
<tr>
<td>upper shadow</td>
<td>27</td>
</tr>
</tbody>
</table>

Example 2

An array \( a \) is distributed cyclically onto four nodes. Node \( n(2) \) has its local section of twenty-five 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 \( n(2) \).

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>98</td>
<td>24</td>
</tr>
</tbody>
</table>

Example 3
Since 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 therefore can be accessed in the same manner as a normal coarray.

### 4.8.2 post Construct

**Synopsis**

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

**Syntax**

\[
\begin{align*}
[F] & \quad !$xmp \text{ post ( \texttt{nodes-ref, tag} )} \\
[C] & \quad \#pragma \text{ xmp post ( \texttt{nodes-ref, tag} )}
\end{align*}
\]

**Description**

This construct ensures that the execution of statements that precede it is completed before statements that follow the matching wait construct start to be 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**

- \texttt{nodes-ref} must represent one node.
- \texttt{tag} must be an expression of type default integer, in XcalableMP Fortran, or type \texttt{int}, in XcalableMP C.
Example

Example 1

XcalableMP Fortran

S1

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

XcalableMP Fortran

S2

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

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 1. 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 matches this wait construct because both nodes-ref and tag are omitted in this wait construct.

4.8.3 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 this construct 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 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 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 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.
4.8.4 [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, a 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, a node does not stop at the lock construct and the variable success is defined with the value false and 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 following symbolic constants when executing lock/unlock construct.

- XMP_STAT_SUCCESS
- XMP_STAT_LOCKED
- XMP_STAT_UNLOCKED
- XMP_STAT_LOCKED_OTHER_IMAGE

If an execution of lock/unlock construct is success, the status is defined with XMP_STAT_SUCCESS. A 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 execution of these constructs, the status is defined with a value which is different from the value of XMP_STAT_SUCCESS, XMP_STAT_LOCKED, XMP_STAT_UNLOCKED, and XMP_STAT_LOCKED_OTHER_IMAGE.
Example

```
#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){
  }
#pragma xmp unlock(lock_obj[2])
```
Chapter 5

Base Language Extensions in XcalableMP C

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

5.1 Array Section Notation

Synopsis

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

Syntax

\[[C] \text{array-section} \text{ is } \text{array-name}[\{ \text{triplet} | \text{int-expr} \}] \ldots\]

where triplet is:

\[[base] : [length] [: step] \]

Description

In XcalableMP C, the base language C is extended so that a part of an array, that is, an array section can be put in an array assignment statement, which is described in Section 5.2, 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 be the size of the dimension of the array. 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 determined by all possible subscript lists specified by the sequence of triplet’s 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,

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

some array sections can be specified as follows:

- `A[:]` the whole of A

5.2 Array Assignment Statement

Synopsis

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

Syntax

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

Description

The value of each element of the result of the rhs expression is assigned to the corresponding element of the lhs array section. When an operator or an elemental function (see section 7.7) is applied to array sections in the rhs expression, it is evaluated to an array section that has the same shape as of the operands or arguments and each element of which is the result of the operator or function applied to corresponding element of the operands or arguments, where a scalar object is assumed to be an array section that has the same shape as of the array section(s) and each element of which 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 rhs expression and the lhs must have the same shape, i.e., the same number of dimensions and size of each dimension.
- [C] If `array-section` on the lhs is followed by ":[int-expr]...", it must be a coarray.
Examples


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

### 5.3 Built-in Functions for Array Section

Some built-in functions are defined that can accept one or more array sections as arguments, and, 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.

Each of the built-in functions for array section are described in Sections 7.7 and 7.8.

### 5.4 Pointer to Global Data

#### 5.4.1 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 appeared in an expression is evaluated to the pointer to the base address of its local section on each node. The pointer, as a normal (local) pointer, can be operated on each node.

#### 5.4.2 The 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, as a normal (local) pointer, can be operated on the node.

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

### 5.5 Dynamic Allocation of Global Data

In XcalableMP C, it is possible to allocate global arrays at runtime only when they are one-dimensional. Such allocation is done through 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 `xmp_malloc` library procedure and assign the result value to the pointer on each node.

The specification of `xmp_malloc` is described in section 7.4.1.
Example

A pointer `pa` to a float is declared in line 5 and aligned with a template `t` in line 6. `t` is initially undefined and fixed by the `template_fix` directive in line 10. The storage for a global data, that is, each of its local section is allocated with `xmp_malloc` and `pa` is set to point it on each node in line 12. For details of the operator `xmp_desc_of`, refer to the next section.

```c
#pragma nodes p(NP)
#pragma xmp template t(:)
#pragma xmp distribute t(block) onto p

5 float *pa;
#pragma xmp align pa[i] with t(i)
...
#pragma xmp template_fix t(N)

pa = (float *)xmp_malloc(xmp_desc_of(pa), N);
```

5.6 The 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, for example, as an argument of some inquiry procedures. The type of the result, `xmp_desc_t`, is implementation-dependent, and 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 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 a few restrictions as possible.

6.1 General Rule

In XcalableMP, a procedure invocation itself is 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 of 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 is explained below.

A 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 by the compiler of the base language. This rule makes it possible that a local actual argument in a procedure written in XcalableMP can be associated with a dummy argument of a procedure written in the base language.

If both of an actual and its associating dummy arguments 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 that the backend compiler of the base language can treat them in its usual way.

The rest of this chapter specifies how global data appearing as an actual and a 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 two conditions:
– its subscript list is a list of zero or more colons ("::") followed by zero or more int-expr’s;
– a subscript of the dimension having 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 a global dummy that is an explicit-shape or assumed-size array, and the other is descriptor association, which is for all other global dummy.

### 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. Also, if the actual argument is an element of a global data, 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 the dummy argument if the dummy argument is an explicit-shape or assumed-size array. According to this (extended) sequence association rule, 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 therefore is 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 a 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 as to be 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 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.

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

```fortran
subroutine xmp_sub1
!$xmp nodes p(4)
!$xmp template t(100)
!$xmp distribute t(block) onto p
real a(100)
```
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 applied also in this case. The caller subroutine \( \text{xmp\_sub1} \) passes the base address of the local section of \( a \) on each node, and the callee \( \text{f\_sub2} \) receives it and initializes \( x \) with the storage starting from it (Figure 6.2).
Example 3 The actual argument \( a(:,1) \) is a contiguous section 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 section. Hence, \( f_{\text{sub2}} \) is invoked only by \( p(1) \) (Figure 6.2).

```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 6.2: Sequence Association with a Local Dummy Argument
6.2. ARGMUMENT PASSING MECHANISM IN XCALABLEMP FORTRAN

6.2.1 Argument Passing Mechanism

**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 6.3).

```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
```

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

**Example 5** Even if either the global actual or dummy argument has a full shadow, the sequence association rule 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 the local section of `x` starts from the received address (Figure 6.3).

6.2.2 Descriptor Association of Global Data

When the actual argument is a global data 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.
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 hold information on a global data (see \ref{sec:global-data}), if the dummy is a global data; 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.

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 the 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 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).

```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
```

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).
6.3 Argument Passing Mechanism in XcalableMP C

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

**Implementation.** The name of a global data appearing as an actual argument is treated by the XcalableMP compiler as the pointer to the first element of its local section on each node.
6.3. ARGUMENT PASSING MECHANISM IN XCALABLEMP C

On a node onto which no part of the global data is mapped, the pointer is set to an unspecified value (e.g. null). Note that an element of a global data 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 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 as to be composed of the local sections on the executing nodes.

Such 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 section starts from the address held by the received pointer (Figure 6.8).

```c
void xmp_func1()
{
    #pragma xmp nodes p(4)
    #pragma xmp template t(0:99)
    #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(0:99)
    #pragma xmp distribute t(block) onto p
```
Example 2  The global actual argument \texttt{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 \texttt{x} on each node starts from the address held by the received pointer (Figure 6.9).

Example 3  The actual argument \texttt{a[0]} is an element of the global data and the dummy argument \texttt{x} is a scalar, in which case the normal argument-passing rule of C for variables of a basic type (i.e. “pass-by-value”) is applied. However, only the node \texttt{p(1)} owns the element. Hence, \texttt{c_func2} is invoked only by \texttt{p(1)} (Figure 6.10).
6.3. ARGUMENT PASSING MECHANISM IN XCALABLEMP C

Figure 6.9: Passing to a Local Dummy Argument

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

//pragma xmp task on p(1)
c_func2(a[0]);
```

Figure 6.10: Passing an Element of a Global Data as an Actual Argument to a Local Dummy Argument

```c
void c_func2(float x)
{
...
```
Chapter 7

Intrinsic and Library Procedures

This specification defines various procedures for system inquiry, synchronization, computations, etc. The procedures are provided as intrinsic procedures in XcalableMP Fortran and library procedures in XcalableMP C.

7.1 [F] 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 5.6.

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 the descriptor, type(xmp_desc), is implementation-dependent, and 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
- xmp_all_num_nodes
- xmp_node_num
- xmp_num_nodes
- 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 primary node set, of the node that calls xmp_all_node_num.

Arguments

none.

7.2.2 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.

7.2.3 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.

7.2.4 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.
7.3. SYNCHRONIZATION FUNCTIONS

Arguments

none.

7.2.5 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.

7.2.6 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 equal to the number of seconds between successive clock ticks.

Arguments

none.

7.3 Synchronization Functions

7.3.1 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.

7.4 Memory Allocation Functions

7.4.1 [C] `xmp_malloc`

```c
void* xmp_malloc(xmp_desc_t d, size_t size)
```

Synopsis

The `xmp_malloc` routine allocates a storage for the local section of a one-dimensional global array of size `size` 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 7.3.

Arguments

- `d` is the descriptor associated with a pointer to the one-dimensional global array to be allocated.
- `size` is the size of the global array to be allocated.

7.5 Mapping Inquiry Functions

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

7.5.1 `xmp_nodes_ndims`

Format

[F] `integer function xmp_nodes_ndims(d, ndims)`

```c
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`. 
7.5. MAPPING INQUIRY FUNCTIONS

7.5.2 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.

7.5.3 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 an extent of the target dimension of the node array specified by d.
7.5.4 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_PRIMARY_NODES (Primary nodes)
- XMP_EQUIVALENCE_NODES (Equivalence nodes)

These are named constants defined in module xmp_lib and in include file xmp_lib.h in XcalableMP Fortran, and symbolic constants defined in 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.

7.5.5 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 and a subscript list that represent a node set that is assigned to the target node array in the nodes directive. This function returns with 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.

### 7.5.6 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 which 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-dependent non-zero integer value in XcalableMP C if the template specified by **d** is fixed; otherwise to false in XcalableMP Fortran and zero in XcalableMP C.

### 7.5.7 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

- \( \text{ndims} \) is the rank of the template specified by \( d \).

7.5.8 \( \text{xmp\_template\_lbound} \)

Format

\[\text{[F]} \quad \text{integer function} \quad \text{xmp\_template\_lbound}(d, \text{dim}, \text{lbound})\]

\[\text{type(xmp\_desc)} \quad d\]

\[\text{integer} \quad \text{dim}\]

\[\text{integer} \quad \text{lbound}\]

\[\text{[C]} \quad \text{int} \quad \text{xmp\_template\_lbound}(\text{xmp\_desc\_t} \quad d, \text{int} \quad \text{dim}, \text{int} \quad *\text{lbound})\]

Synopsis

The \( \text{xmp\_template\_lbound} \) function provides the lower bound of each dimension of the template. This function returns with failure when the lower bound is not fixed.

Input Arguments

- \( d \) is a descriptor of a template.
- \( \text{dim} \) is the target dimension of the template.

Output Arguments

- \( \text{lbound} \) 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.

7.5.9 \( \text{xmp\_template\_ubound} \)

Format

\[\text{[F]} \quad \text{integer function} \quad \text{xmp\_template\_ubound}(d, \text{dim}, \text{ubound})\]

\[\text{type(xmp\_desc)} \quad d\]

\[\text{integer} \quad \text{dim}\]

\[\text{integer} \quad \text{ubound}\]

\[\text{[C]} \quad \text{int} \quad \text{xmp\_template\_ubound}(\text{xmp\_desc\_t} \quad d, \text{int} \quad \text{dim}, \text{int} \quad *\text{ubound})\]

Synopsis

The \( \text{xmp\_template\_ubound} \) function provides the upper bound of each dimension of the template. This function returns with failure when the upper bound is not fixed.

Input Arguments

- \( d \) is a descriptor of a template.
- \( \text{dim} \) is the target dimension of the template.
7.5. MAPPING INQUIRY FUNCTIONS

Output Arguments

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

7.5.10  **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**.

7.5.11  **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 \( d \).

### 7.5.12 xmp_dist_gblockmap

**Format**

\[
[F] \quad \text{integer function } \ xmp\_dist\_gblockmap(d, \ dim, \ map) \\
\text{type(xmp\_desc)} \quad \text{d} \\
\text{integer} \quad \text{dim} \\
\text{integer} \quad \text{map(N)} \\
[C] \quad \text{int} \quad \text{xmp\_dist\_gblockmap(xmp\_desc\_t d, int dim, int map[])}
\]

**Synopsis**

The **xmp_dist_gblockmap** function provides the mapping array of the **gblock** distribution.

When \( \text{dim} \) dimension of the global array is distributed by **gblock** and its mapping array is fixed, this function returns zero; otherwise it returns an implementation-dependent negative integer value.

**Input Arguments**

- \( d \) is a descriptor of a template.
- \( \text{dim} \) is the target dimension of the template.

**Output Arguments**

- \( \text{map} \) 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 \( \text{map} \) is set to the value of the i-th element of the target mapping array.

### 7.5.13 xmp_dist_nodes

**Format**

\[
[F] \quad \text{integer function } \ xmp\_dist\_nodes(d, \ dn) \\
\text{type(xmp\_desc)} \quad \text{d} \\
\text{type(xmp\_desc)} \quad \text{dn} \\
[C] \quad \text{int} \quad \text{xmp\_dist\_nodes(xmp\_desc\_t d, xmp\_desc\_t *dn)}
\]

**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**

- \( \text{dn} \) is the descriptor of the node array.
7.5.14  \texttt{xmp\_dist\_axis}

\textbf{Format}

\begin{verbatim}
[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)
\end{verbatim}

\textbf{Synopsis}

The \texttt{xmp\_dist\_axis} function provides the dimension of the node array onto which a dimension of a template is distributed. This function returns with failure when the dimension of the template is not distributed.

\textbf{Input Arguments}

- \texttt{d} is a descriptor of a template.
- \texttt{dim} is the target dimension of the template.

\textbf{Output Arguments}

- \texttt{axis} is a dimension of the node array onto which the target dimension of the template specified by \texttt{d} is distributed. When the dimension of the template is not distributed, it is set to undefined.

7.5.15  \texttt{xmp\_align\_axis}

\textbf{Format}

\begin{verbatim}
[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)
\end{verbatim}

\textbf{Synopsis}

The \texttt{xmp\_align\_axis} function provides the dimension of the template with which a dimension of a global array is aligned. This function returns with failure when the dimension of the global array is not aligned.

\textbf{Input Arguments}

- \texttt{d} is a descriptor of a global array.
- \texttt{dim} is the target dimension of the global array.

\textbf{Output Arguments}

- \texttt{axis} is the dimension of the template with which the target dimension of the global array specified by \texttt{d} is aligned. When the dimension of the global array is not aligned, or collapsed, it is set to undefined.
7.5.16  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 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.

7.5.17  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 which shows whether the dimension of the template with which a global array is aligned is replicated or not.

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.
7.5. MAPPING INQUIRY FUNCTIONS

7.5.18  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.

7.5.19  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 \( d \).

7.5.20  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 \texttt{xmp\_array\_lshadow} function provides the size of lower shadow of a dimension of a global array.

Input Arguments
- \texttt{d} is a descriptor of a global array.
- \texttt{dim} is the target dimension of the global array.

Output Arguments
- \texttt{lshadow} is the size of the lower shadow of the target dimension of the global array specified by \texttt{d}.

7.5.21 \texttt{xmp\_array\_ushadow}

Format

\begin{verbatim}
[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)
\end{verbatim}

Synopsis
The \texttt{xmp\_array\_ushadow} function provides the size of upper shadow of a dimension of a global array.

Input Arguments
- \texttt{d} is a descriptor of a global array.
- \texttt{dim} is the target dimension of the global array.

Output Arguments
- \texttt{ushadow} is the size of the upper shadow of the target dimension of the global array specified by \texttt{d}.

7.5.22 \texttt{xmp\_array\_lbound}

Format

\begin{verbatim}
[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)
\end{verbatim}

Synopsis
The \texttt{xmp\_array\_lbound} function provides the lower bound of a dimension of a global array. This function returns with failure when the lower bound is not fixed.
Input Arguments

- \( \text{d} \) is a descriptor of a global array.
- \( \text{dim} \) is the target dimension of the global array.

Output Arguments

- \( \text{lbound} \) is the lower bound of the target dimension of the global array specified by \( \text{d} \). When the lower bound is not fixed, it is set to undefined.

7.5.23 xmp_array_ubound

Format

\[
\begin{align*}
\text{[F]} & \quad \text{integer function} \quad \text{xmp_array_ubound}(d, \text{dim, ubound}) \\
\text{type(xmp_desc)} & \quad \text{d} \\
\text{integer} & \quad \text{dim} \\
\text{integer} & \quad \text{ubound} \\
\text{[C]} & \quad \text{int} \quad \text{xmp_array_ubound(xmp_desc_t d, int dim, int *ubound)}
\end{align*}
\]

Synopsis

The xmp_array_ubound function provides the upper bound of a dimension of a global array. This function returns with failure when the upper bound is not fixed.

Input Arguments

- \( \text{d} \) is a descriptor of a global array.
- \( \text{dim} \) is the target dimension of the global array.

Output Arguments

- \( \text{ubound} \) is the upper bound of the target dimension of the global array specified by \( \text{d} \). When the upper bound is not fixed, it is set to undefined.

7.6 [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 is of 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 not defined how the transformational functions with a global array or its subobject being an argument work. 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 7.8 as alternatives to these transformational functions.

7.7 [C] Built-in Elemental Functions

Some built-in elemental functions that could 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 of the same shape and mapping as the argument. The values of the elements of the result are the same as 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 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.

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.

Table 7.1: Built-in Elemental Functions in XcalableMP C (The first line means the element type of their argument(s) and return value.)

<table>
<thead>
<tr>
<th></th>
<th>double</th>
<th>float</th>
<th>long double</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>acosf</td>
<td>acosl</td>
<td></td>
</tr>
<tr>
<td>asin</td>
<td>asinf</td>
<td>asinl</td>
<td></td>
</tr>
<tr>
<td>atan</td>
<td>atanf</td>
<td>atanl</td>
<td></td>
</tr>
<tr>
<td>atan2</td>
<td>atan2f</td>
<td>atan2l</td>
<td></td>
</tr>
<tr>
<td>cos</td>
<td>cosf</td>
<td>cosl</td>
<td></td>
</tr>
<tr>
<td>sin</td>
<td>sinf</td>
<td>sinl</td>
<td></td>
</tr>
<tr>
<td>tan</td>
<td>tanf</td>
<td>tanl</td>
<td></td>
</tr>
<tr>
<td>cosh</td>
<td>coshf</td>
<td>coshl</td>
<td></td>
</tr>
<tr>
<td>sinh</td>
<td>sinhf</td>
<td>sinhl</td>
<td></td>
</tr>
<tr>
<td>tanh</td>
<td>tanhf</td>
<td>tanhl</td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td>expf</td>
<td>expl</td>
<td></td>
</tr>
<tr>
<td>frexp</td>
<td>frexpf</td>
<td>frexpl</td>
<td></td>
</tr>
<tr>
<td>ldexp</td>
<td>ldexpf</td>
<td>ldexpl</td>
<td></td>
</tr>
<tr>
<td>log</td>
<td>logf</td>
<td>logl</td>
<td></td>
</tr>
<tr>
<td>log10</td>
<td>log10f</td>
<td>log10l</td>
<td></td>
</tr>
<tr>
<td>fabs</td>
<td>fabsf</td>
<td>fabsl</td>
<td></td>
</tr>
<tr>
<td>pow</td>
<td>powf</td>
<td>powl</td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>sqrtf</td>
<td>sqrtl</td>
<td></td>
</tr>
<tr>
<td>ceil</td>
<td>ceilf</td>
<td>ceill</td>
<td></td>
</tr>
<tr>
<td>floor</td>
<td>floorf</td>
<td>floorl</td>
<td></td>
</tr>
<tr>
<td>fmod</td>
<td>fmodf</td>
<td>fmodl</td>
<td></td>
</tr>
</tbody>
</table>
7.8 Intrinsic/Built-in Transformational Procedures

Some intrinsic/built-in transformational procedures are defined for non-elemental operation 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 whole of the array.

7.8.1 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 determined by vectors idx1, ..., idxn.

This procedure produces the same result as the following Fortran assignment statement does when x, a, and idx1, ..., idxn are not mapped.

\[ x(:, :, \ldots, :) = a(idx1(:, :, \ldots), \ldots, idxn(:, :, \ldots)) \]

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.

7.8.2 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 does when x, a, and idx1, ..., idxn are not mapped.

\[ x(:, :, \ldots) = a(idx1(:, :, \ldots), \ldots, idxn(:, :, \ldots)) \]

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.
- \(\text{idx}_1, ..., \text{idx}_n\) are integer arrays of the same shape and mapping as \(x\). The number of \(\text{idx}\)'s is equal to the rank of \(a\).

7.8.3 \texttt{xmp\_pack}

Format

\[
\begin{align*}
[F] & \quad \texttt{xmp\_pack}(v, a, [\text{mask}]) \\
[C] & \quad \texttt{void xmp\_pack}(v[:], a[:], ..., [\text{mask}[:]])
\end{align*}
\]

Synopsis

The \texttt{xmp\_pack} procedure packs all the elements of an array \(a\), if \texttt{mask} is not specified, or the elements selected by \texttt{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 of \(v\) and any shape and mapping.
- (optional) \(\text{mask}\) is a logical array of the same shape and mapping as \(a\).

7.8.4 \texttt{xmp\_unpack}

Format

\[
\begin{align*}
[F] & \quad \texttt{xmp\_unpack}(a, v, [\text{mask}]) \\
[C] & \quad \texttt{void xmp\_unpack}(a[:], v[:], [\text{mask}[:]])
\end{align*}
\]

Synopsis

The \texttt{xmp\_unpack} procedure unpacks a vector \(v\) to all the elements of an array \(a\), if \texttt{mask} is not specified, or the elements selected by a mask \texttt{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) \(\text{mask}\) is a logical array of the same shape and mapping as \(a\).
7.8.5 **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.

7.8.6 **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 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.

7.8.7 **xmp_sort_up**

**Format**

[F] xmp_sort_up(v1, v2)
[C] void xmp_sort_up(v1[:], v2[:])
Synopsis
The \texttt{xmp\_sort\_up} procedure sets the result obtained by sorting elements of a vector \texttt{v2} in ascending order to a vector \texttt{v1}.

Output Arguments
- \texttt{v1} is a one-dimensional array of any numerical type, shape and mapping.

Input Arguments
- \texttt{v2} is a one-dimensional array of the same type, shape and mapping as \texttt{v1}.

7.8.8 \texttt{xmp\_sort\_down}

Format

\begin{verbatim}
[F] xmp_sort_down(v1, v2)
[C] void xmp_sort_down(v1[:], v2[:])
\end{verbatim}

Synopsis
The \texttt{xmp\_sort\_down} procedure sets the result obtained by sorting elements of a vector \texttt{v2} in descending order to a vector \texttt{v1}.

Output Arguments
- \texttt{v1} is a one-dimensional array of any numerical type, shape and mapping.

Input Arguments
- \texttt{v2} is a one-dimensional array of the same type, shape and mapping as \texttt{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 therefore may be placed in the sequential part, or one of the single, master, and critical regions that is closely nested inside a parallel region whose parent thread is the initial thread;

- with the exception that XcalableMP’s loop directive that controls a loop can be placed immediately inside 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 implementation-dependent.

Examples

Assume that the following codes are placed in the sequential part of the program.

```c
#pragma omp parallel for
for (...){
    #pragma xmp barrier // NG because not single-threaded
}
```

```c
#pragma omp parallel for
for (...){
    #pragma omp single
    {
        #pragma xmp barrier // OK because single-threaded
          // (inside a single region)
    }
}
```

```c
#pragma omp parallel for
#pragma xmp loop // OK because immediately nested
for (...){
    ...
}
```

105
#pragma xmp loop  // OK because single-threaded (not nested)
#pragma omp parallel for
for (...){
   ...
}

#pragma xmp loop  // OK because single-threaded (not nested)
for (...){
   #pragma omp parallel for
   for (...) { ... }
}

#pragma omp parallel for
for (...){
   #pragma xmp loop  // NG because not immediately nested
   for (...) { ... }
}
Bibliography


Appendix A

Programming Interface for MPI

This chapter describes the programming interface for MPI, which are widely used for parallel programming for 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

```
#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(2:3)
{
  MPI_Comm comm = xmp_get_mpi_comm(); // get the MPI communicator of p(2:3)

  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 Design of the Interface

A 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 through an Interface Procedure](image)

- When the numerical library routine needs information on a global array, the interface extracts it from the descriptor using some query routines provided by XcalableMP and passes it to the numerical library routine as arguments.

- 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-dependent, are shown. Specifications of the functions below are from the Omni XcalableMP compiler [http://www.xcalablemp.org/download.html].
B.2.1 xmp_array_gtol

[F] integer function xmp_array_gtol(d, g_idx, l_idx)
  type(xmp_desc) d
  integer g_idx(NDIMS)
  integer l_idx(NDIMS)

[C] void xmp_array_gtol(xmp_desc_t d, int g_idx[], int l_idx[])

Synopsis

The xmp_array_gtol function translates an index (specified by g_idx) of a global array (specified by d) into the corresponding index of its local section and sets 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 a global array.
- [F] g_idx is a rank-one integer array of the size equal to the rank of the target global array specified by d. NDIMS is the rank of the target global array.
- [C] g_idx is a one-dimensional integer array.

Output Arguments

- [F] l_idx is a rank-one integer array of the size equal to the rank of the target global array specified by d. NDIMS is the rank of the target global array.
- [C] l_idx is a one-dimensional integer 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 includes 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 Arguments

- 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_lead_dim*

**Format**

[F] `integer function xmp_array_lead_dim(d, size)`

```fortran
  type(xmp_desc) d
  integer size(N)
```

[C] `int xmp_array_lead_dim(xmp_desc_t d, int size[])`

**Synopsis**

The `xmp_array_lead_dim` function provides the leading dimension of each local section of the target global array.

**Input Arguments**

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

**Output Arguments**

- `size` is a one-dimensional integer array the extent of which must be more than or equal to the rank 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 depends on BLACS (Basic Linear Algebraic Communication Sub-programs). 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`. 
APPENDIX B. INTERFACE TO NUMERICAL LIBRARIES

```fortran
__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
```
Example 2  This example shows an XcalableMP procedure using the interface of Example 1.

```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,npro=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,npro,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 \[\text{http://www.xcalablemp.org/download.html}\] is presented for reference.

The XcalableMP specification says that 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 of 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 sequence can be maintained, with 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, as a result of the model above, memory usage can be non-uniform among the nodes.

Example

```
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 \((\text{block,block})\) 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 the way to use I/O statements and standard I/O functions in the base languages, in which 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 in local 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] A 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] Any array section of a global array cannot be referred to as an I/O item.

D.1.2 Master I/O

Master I/O is input and output for the file that corresponds to an executing node set. Master I/O is collective execution.

In master I/O, a global data 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 among the executing node set arbitrarily by the system, 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 by the base language, and vice versa.

D.1.3 Global I/O

Global I/O is input and output for the file that corresponds to an executing node set. Some executions of global I/O are collective and the others are independent. In a large system with
Table D.1: Global I/O

<table>
<thead>
<tr>
<th></th>
<th>independent/collection</th>
<th>access method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collective I/O</td>
<td>collective</td>
<td>sequential access</td>
</tr>
<tr>
<td>Atomic I/O</td>
<td>independent</td>
<td>sequential access</td>
</tr>
<tr>
<td>Direct I/O</td>
<td>independent</td>
<td>direct access</td>
</tr>
</tbody>
</table>

many nodes, global I/O can be expected higher speed and less memory consumption execution than master I/O.

[F] It is provided in the form of directives for a part of I/O statements, such as OPEN, CLOSE, READ and WRITE statements.

[C] It is provided in the form of service functions and the 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 the file header and footer particular to the base language.

There are three kinds of global I/O, as shown in Table D.1. Collective global I/O is collective execution and sequential file access. It handles global data in the sequential order, similar to master I/O. Atomic global I/O is 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 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 focusing on a node since local I/O is executed on each node individually.

Master I/O is a directive extension, in which the execution result matches the one 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. It avoids 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 file connections, dynamic connection and preconnection. Dynamic connection connects a file during execution of the program. Preconnection connects a file at the
beginning of execution of the program and therefore it can 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 dependent which nodes can access the standard input, output and error files. It is also implementation dependent how the accesses to the files with the same name by multiple nodes behaves. 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 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 termination of the program.

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 and CLOSE statements.

D.2.3 File Connection in Global I/O

Dynamic connection of global I/O is collective execution and is valid for the executing node set. Global I/O files cannot be preconnected.

[F]

An OPEN statement 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 termination of the program.

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 termination of 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 string-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 allowed as master I/O.

Description

An I/O statement specified with master I/O directive accesses a file whose format is the same as the one 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 where the file was connected.

A master node, 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

<table>
<thead>
<tr>
<th>I/O item</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>input item</td>
<td>name of global array</td>
</tr>
<tr>
<td></td>
<td>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 by the size of data.</td>
</tr>
<tr>
<td>array element of global array</td>
<td>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 by the size of data.</td>
</tr>
<tr>
<td>local variable</td>
<td>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 by the size of data.</td>
</tr>
<tr>
<td>implied DO loop</td>
<td>For each input item, repeat the above operation.</td>
</tr>
<tr>
<td>output item</td>
<td>name of global array</td>
</tr>
<tr>
<td></td>
<td>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 by the size of data.</td>
</tr>
<tr>
<td>array element of global array</td>
<td>The element of the global array is written to the file. A file position increases by the size of data.</td>
</tr>
<tr>
<td>local variable and expression</td>
<td>The value evaluated on the master node is written to the file. The file positioning increases by the size of data.</td>
</tr>
<tr>
<td>implied DO loop</td>
<td>For each output item, repeat the above operation.</td>
</tr>
</tbody>
</table>

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. And 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 specified with ERR, END or EOR specifier is satisfied, all nodes of 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 be higher performance and lower memory consumption than master I/O. The file format is compatible with the one in MPI-IO.

Global I/O consists of three kinds, 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 available for an io-statement are shown in the following table. Definition of each specifier is described in the specification of the base language.

Case of global_io construct without a direct clause:

<table>
<thead>
<tr>
<th>I/O statement</th>
<th>available specifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>UNIT, IOSTAT, FILE, STATUS, POSITION, ACTION, ACCESS, FORM</td>
</tr>
<tr>
<td>CLOSE</td>
<td>UNIT, IOSTAT, STATUS</td>
</tr>
<tr>
<td>READ</td>
<td>UNIT, IOSTAT</td>
</tr>
<tr>
<td>WRITE</td>
<td>UNIT, IOSTAT</td>
</tr>
</tbody>
</table>

Case of global_io construct with a direct clause:

<table>
<thead>
<tr>
<th>I/O statement</th>
<th>available specifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPEN</td>
<td>UNIT, IOSTAT, FILE, STATUS, RECL, ACTION, ACCESS, FORM</td>
</tr>
<tr>
<td>CLOSE</td>
<td>UNIT, IOSTAT, STATUS</td>
</tr>
<tr>
<td>READ</td>
<td>UNIT, REC, IOSTAT</td>
</tr>
<tr>
<td>WRITE</td>
<td>UNIT, REC, IOSTAT</td>
</tr>
</tbody>
</table>

An input item and an 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 explicitly be connected by the OPEN statement specified with a global_io directive.

The OPEN statement 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 reopen 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 specified with a `global_io` directive, otherwise the result of I/O is not guaranteed. The CLOSE statement 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**

<table>
<thead>
<tr>
<th>specifiers</th>
<th>value</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT</td>
<td>external file unit (scalar constant expression)</td>
<td>not omissible</td>
</tr>
<tr>
<td>FILE</td>
<td>file name (scalar CHARACTER expression)</td>
<td>not omissible</td>
</tr>
<tr>
<td>STATUS</td>
<td>'OLD', 'NEW', 'REPLACE' or 'UNKNOWN'</td>
<td>'UNKNOWN'</td>
</tr>
<tr>
<td>POSITION</td>
<td>'ASIS', 'REWIND' or 'APPEND'</td>
<td>'ASIS'</td>
</tr>
<tr>
<td>ACTION</td>
<td>'READ', 'WRITE' or 'READ-WRITE'</td>
<td>processor dependent</td>
</tr>
<tr>
<td>RECL</td>
<td>the value of the record length</td>
<td>not omissible</td>
</tr>
<tr>
<td>ACCESS</td>
<td>'SEQUENTIAL' or 'DIRECT'</td>
<td>'SEQUENTIAL'</td>
</tr>
<tr>
<td>FORM</td>
<td>'FORMATTED' or 'UNFORMATTED'</td>
<td>For direct access, UNFORMATTED. For sequential access, this specifier shall not be omitted.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>specifiers</th>
<th>value</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT</td>
<td>external file unit (scalar constant expression)</td>
<td>not omissible</td>
</tr>
<tr>
<td>STATUS</td>
<td>'KEEP' or 'DELETE'</td>
<td>'KEEP'</td>
</tr>
</tbody>
</table>

- **READ/WRITE statement**

<table>
<thead>
<tr>
<th>specifiers</th>
<th>value</th>
<th>default</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT</td>
<td>external file unit (scalar constant expression)</td>
<td>not omissible</td>
</tr>
<tr>
<td>REC</td>
<td>the value of the number of record (scalar constant expression)</td>
<td>not omissible</td>
</tr>
</tbody>
</table>
REC is available only if the directive has a direct clause.

- When a scalar variable of default INTEGER is specified to IOSTAT, an error code is set to the specifiers.

OPEN, CLOSE, READ and WRITE statements specified with `global_io` directives without atomic and direct clauses are called collective OPEN, collective CLOSE, collective READ, and collective WRITE statements respectively. These all statements are called collective I/O statements.

OPEN, CLOSE, READ and WRITE statements specified with `global_io` directives with atomic clauses are called atomic OPEN, atomic CLOSE, atomic READ, and atomic WRITE statements respectively. These all 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. These all statements are called direct I/O statements.

The file connected by a collective, atomic or direct OPEN statement can be read/be written only by the same type of READ/WRITE statements. The file can be disconnected by the same type of a 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 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

```plaintext
!$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 can differ every execution time even for the same program.

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
<table>
<thead>
<tr>
<th>I/O item</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>input item</td>
<td>name of global array</td>
</tr>
<tr>
<td></td>
<td>local variable</td>
</tr>
<tr>
<td>output item</td>
<td>name of global array</td>
</tr>
<tr>
<td></td>
<td>local variable, expression</td>
</tr>
</tbody>
</table>

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. And then, the file positioning seeks in every READ and WRITE statement by the length of the input/output data.

### D.4.4 Direct Global I/O Statement

Direct I/O statements read/write shared files with specification of 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 independently. The file positioning is shifted from the top of the file by 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 dependent.

### 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 to read/write data 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 following table. The `xmp_allocate_range()` function is used to allocate memory. The `xmp_set_range()` function is used to set ranges of a array section. The `xmp_free_range()` function releases the memory for the descriptor.
<table>
<thead>
<tr>
<th>function name</th>
<th>xmp_range_t *xmp_allocate_range(n_dim)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>int n_dim</td>
</tr>
<tr>
<td></td>
<td>the number of dimensions</td>
</tr>
<tr>
<td>return value</td>
<td>xmp_range_t*</td>
</tr>
<tr>
<td></td>
<td>descriptor of array section. NULL is returned when a program abend.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>function name</th>
<th>void xmp_set_range(rp, i_dim, lb, length, step)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_range_t *rp</td>
</tr>
<tr>
<td></td>
<td>descriptor</td>
</tr>
<tr>
<td></td>
<td>int i_dim</td>
</tr>
<tr>
<td></td>
<td>target dimension</td>
</tr>
<tr>
<td></td>
<td>int lb</td>
</tr>
<tr>
<td></td>
<td>lower bound of array section in the dimension i_dim</td>
</tr>
<tr>
<td></td>
<td>int length</td>
</tr>
<tr>
<td></td>
<td>length of array section in the dimension i_dim</td>
</tr>
<tr>
<td></td>
<td>int step</td>
</tr>
<tr>
<td></td>
<td>stride of array section in the dimension i_dim</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>function name</th>
<th>xmp_file_t *xmp_fopen_all(fname, amode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>const char *fname file name</td>
</tr>
<tr>
<td></td>
<td>const char *amode equivalent to fopen of POSIX, combination of “rwa+”</td>
</tr>
<tr>
<td>return value</td>
<td>xmp_file_t* file structure. NULL is returned when a program abend.</td>
</tr>
</tbody>
</table>

File view is initialized, where file view is based on the MPI-IO vile view mechanism. The value of shared and individual file pointers depend on the value of amode.

<table>
<thead>
<tr>
<th>amode</th>
<th>intended purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Open for reading only. File pointer points the beginning of the file.</td>
</tr>
<tr>
<td>r+</td>
<td>Open an existing file for update (reading and writing). File pointer points the beginning of the file.</td>
</tr>
<tr>
<td>w</td>
<td>Create for writing. If a file by that name already exists, it will be overwritten. File pointer points the beginning of the file.</td>
</tr>
<tr>
<td>w+</td>
<td>Create a new file for update (reading and writing). If a file by that name already exists, it will be overwritten. File pointer points the beginning of the file.</td>
</tr>
<tr>
<td>a</td>
<td>Append; open for writing at end-of-file or create for writing if the file does not exist. File pointer points the end of the file.</td>
</tr>
<tr>
<td>a+</td>
<td>Open for append; open (or create if the file does not exist) for update at the end of the file. File pointer points the beginning of the file.</td>
</tr>
</tbody>
</table>

D.5.1.2 xmp_fclose_all

xmp_fclose_all closes a global I/O file. Collective execution.

<table>
<thead>
<tr>
<th>function name</th>
<th>int *xmp_fclose_all(fh)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh file structure</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>0: normal termination</td>
</tr>
<tr>
<td></td>
<td>1: abnormal termination, fh is NULL.</td>
</tr>
<tr>
<td></td>
<td>2: abnormal termination. error in MPI_File_close.</td>
</tr>
</tbody>
</table>

D.5.1.3 xmp_fseek

xmp_fseek sets the individual file pointer in the file structure. Independent execution.
### D.5.4 xmp_fseek_shared

xmp_fseek_shared sets the shared file pointer in the file structure. Independent execution.

<table>
<thead>
<tr>
<th>function name</th>
<th>int xmp_fseek_shared(fh, offset, whence)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh</td>
</tr>
<tr>
<td></td>
<td>long long offset</td>
</tr>
<tr>
<td></td>
<td>int whence</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
</tr>
</tbody>
</table>

### D.5.5 xmp_ftell

xmp_ftell inquires the position of the individual file pointer in the file structure. Independent execution.

<table>
<thead>
<tr>
<th>function name</th>
<th>long long xmp_ftell(fh)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh</td>
</tr>
<tr>
<td>return value</td>
<td>long long</td>
</tr>
</tbody>
</table>

### D.5.6 xmp_ftell_shared

xmp_ftell_shared inquires the position of shared file pointer in the file structure. Independent execution.
### APPENDIX D. XCALABLEMP I/O

<table>
<thead>
<tr>
<th>function name</th>
<th>long long xmp_ftell_shared(fh)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh</td>
</tr>
<tr>
<td>return value</td>
<td>long long</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>function name</th>
<th>int xmp_file_sync_all(fh)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
</tr>
</tbody>
</table>

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 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 each node owns. 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.

<table>
<thead>
<tr>
<th>function name</th>
<th>int xmp_file_set_view_all(fh, disp, desc, rp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh</td>
</tr>
<tr>
<td></td>
<td>long long</td>
</tr>
<tr>
<td></td>
<td>disp</td>
</tr>
<tr>
<td></td>
<td>xmp_file_t *fh</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>function name</th>
<th>int xmp_file_clear_view_all(fh, disp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh</td>
</tr>
<tr>
<td></td>
<td>long long disp</td>
</tr>
<tr>
<td>return value</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>0: normal termination</td>
</tr>
<tr>
<td></td>
<td>an integer other than 0: abnormal termination</td>
</tr>
</tbody>
</table>

D.5.2.3 xmp_fread_all

xmp_fread_all reads the same data from the position of the shared file pointer onto the all executing nodes. Collective execution.

<table>
<thead>
<tr>
<th>function name</th>
<th>size_t xmp_fread_all(fh, buffer, size, count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh</td>
</tr>
<tr>
<td></td>
<td>void *buffer</td>
</tr>
<tr>
<td></td>
<td>size_t size</td>
</tr>
<tr>
<td></td>
<td>size_t count</td>
</tr>
<tr>
<td>return value</td>
<td>size_t</td>
</tr>
<tr>
<td></td>
<td>Upon successful completion, return the size of read data. Otherwise, negative number shall be returned.</td>
</tr>
</tbody>
</table>

D.5.2.4 xmp_fwrite_all

xmp_fwrite_all writes individual data on the all executing nodes to the position of the shared file pointer. Collective execution.

It is assumed that the file view is set previously. Each node writes its data into its own file view.

<table>
<thead>
<tr>
<th>function name</th>
<th>size_t xmp_fwrite_all(fh, buffer, size, count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh</td>
</tr>
<tr>
<td></td>
<td>void *buffer</td>
</tr>
<tr>
<td></td>
<td>size_t size</td>
</tr>
<tr>
<td></td>
<td>size_t count</td>
</tr>
<tr>
<td>return value</td>
<td>size_t</td>
</tr>
<tr>
<td></td>
<td>Upon successful completion, return the size of written data. Otherwise, negative number shall be returned.</td>
</tr>
</tbody>
</table>

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.
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.

<table>
<thead>
<tr>
<th>function name</th>
<th>size_t xmp_fwrite_darray_all(fh, desc, rp)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td></td>
</tr>
<tr>
<td>xmp_file_t *fh</td>
<td>file structure</td>
</tr>
<tr>
<td>xmp_desc_t desc</td>
<td>descriptor</td>
</tr>
<tr>
<td>xmp_range_t *rp</td>
<td>range descriptor</td>
</tr>
<tr>
<td>return value</td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>Upon successful completion, return the size of read data. Otherwise, negative number shall be returned.</td>
</tr>
</tbody>
</table>

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 by the length of the data.

Before atomic I/O is executed, the file view must be cleared.

[Rationale]
Though the file views must be the same on all processes in order to use the shared file pointer, 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 form the position of the shared file pointer and moves the position by the length of the data. Independently execution.

<table>
<thead>
<tr>
<th>function name</th>
<th>size_t xmp_fread_shared(fh, buffer, size, count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td></td>
</tr>
<tr>
<td>xmp_file_t *fh</td>
<td>file structure</td>
</tr>
<tr>
<td>void *buffer</td>
<td>beginning address of read variables</td>
</tr>
<tr>
<td>size_t size</td>
<td>the size of a read data element</td>
</tr>
<tr>
<td>size_t count</td>
<td>the number of read data elements</td>
</tr>
<tr>
<td>return value</td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>Upon successful completion, return the size of read data. Otherwise, negative number shall be returned.</td>
</tr>
</tbody>
</table>
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 by the length of the data. Independent execution.

<table>
<thead>
<tr>
<th>function name</th>
<th>size_t xmp fwrite_shared(fh, buffer, size, count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh file structure</td>
</tr>
<tr>
<td></td>
<td>void *buffer beginning address of written variables</td>
</tr>
<tr>
<td></td>
<td>size_t size the size of a written data element</td>
</tr>
<tr>
<td></td>
<td>size_t count the number of written data elements</td>
</tr>
<tr>
<td>return value</td>
<td>size_t Upon successful completion, return the size of written data. Otherwise, negative number shall be returned.</td>
</tr>
</tbody>
</table>

D.5.4 Direct Global I/O Functions

Direct I/O is executed independently and using the individual pointer. It individually reads/writes local data from the position of the individual file pointer and moves the position by the length of the data taking account of 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 to set the individual file pointer. It is not recommended using the file view together because of complexity.

D.5.4.1 xmp fread

xmp fread reads data from the position of the individual file pointer and moves the position by the length of the data. Independent execution.

<table>
<thead>
<tr>
<th>function name</th>
<th>size_t xmp fread(fh, buffer, size, count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td>xmp_file_t *fh file structure</td>
</tr>
<tr>
<td></td>
<td>void *buffer beginning address of read variables</td>
</tr>
<tr>
<td></td>
<td>size_t size the size of a read data element</td>
</tr>
<tr>
<td></td>
<td>size_t count the number of read data elements</td>
</tr>
<tr>
<td>return value</td>
<td>size_t Upon successful completion, return the size of read data. Otherwise, negative number shall be returned.</td>
</tr>
</tbody>
</table>

D.5.4.2 xmp fwrite

xmp fwrite writes data to the position of the individual file pointer and moves the position by the length of the data. Independent execution.
## XCALABLEMP I/O

<table>
<thead>
<tr>
<th>function name</th>
<th>size_t xmp_fwrite(fh, buffer, size, count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>argument</td>
<td></td>
</tr>
<tr>
<td>xmp_file_t *fh</td>
<td>file structure</td>
</tr>
<tr>
<td>void *buffer</td>
<td>beginning address of written variables</td>
</tr>
<tr>
<td>size_t size</td>
<td>the size of a written data element</td>
</tr>
<tr>
<td>size_t count</td>
<td>the number of written data elements</td>
</tr>
<tr>
<td>return value</td>
<td></td>
</tr>
<tr>
<td>size_t</td>
<td>Upon successful completion, return the size of written data. Otherwise, negative number shall be returned.</td>
</tr>
</tbody>
</table>
Appendix E

Sample Programs

Example 1

```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++)
```

137
Example 2

XcalableMP C

```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
            #pragma xmp gmove
            a[l][:] = a[k][:];
            #pragma 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
        }
    }
}
```
#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];

        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);
    }
}
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];
    }
}

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;
#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];
}
Index

/perlodic/ modifier, 11

address-of operator, 71
align, 22
align dummy variable, 22
align offset, 22
alignment, 10
array, 45
array assignment in XMP/C, 45
array intrinsic functions, 45
array section in XMP/C, 45
async clause, 45
asynchronous communication, 45

barrier, 45
base language, 7
base program, 7
bcast, 45
block, 45
broadcast variables, 45
built-in elemental functions, 100
built-in functions of XMP/C, 70
built-in transformational procedures, 101

coarray reference, 60
collapse, 45
collective mode (of gmove), 45
combined directive, 45
communication, 10
class, 72
class association, 74
coarray reference, 60
classassociation of operator, 74

Example

align, 22
array, 45
array assignment in XMP/C, 45
array section in XMP/C, 45
async, 45
barrier, 45
bcast, 45
distribute, 45
gmove, 45
local alias, 45
lock, 45
loop, 45
nodes, 18
post, 45
reduction, 45
reflect, 45
shadow, 45
task, 24
tasks, 24
template, 45
unlock, 45
wait, 45
wait async, 45
directives, 7

distribute, 45
distribution, 45
distribution format

* , 45
block, 45
cyclic, 45
gblock, 45
global, 45
gmove, 45

entire node set, 7

data mapping, 3
declarative directive, 3
declarative directives, 13
descriptor, 72
descriptor association, 74
descriptor-of operator, 74

Directive

align, 22
array, 45
async clause, 45
barrier, 45
bcast, 45
distribute, 45
gmove, 45
local alias, 45
lock, 45
loop, 45
nodes, 18
post, 45
reduction, 45
reflect, 45
shadow, 45
task, 24
tasks, 24
template, 45
unlock, 45
wait, 45
wait async, 45
directives, 7
end tasks, 24

gmove, 23

library interface, 11

local_alias, 52

loop, 24, 29, 30, 115

memory-layout, 117

MPI interface, 11

node reference, 17

nodes, 11, 13, 21, 115

OpenMP in XcalableMP programs, 112

post, 27

procedure interface, 73

reduction, 18, 20, 45

reflect, 40

shadow, 25, 40

task, 18, 19, 28, 29, 34, 35

tasks, 24

template, 24, 29, 115

template_fix, 24, 29, 115

wait, 65

wait_async, 39

xmp_desc_of, 39

xmp_malloc, 39

executable directive, 8

executable directives, 13

executing node, 9

executing node array, 9

executing node set, 9, 9

full shadow, 25

gblock, 23

global, 8

global actual argument, 27

global communication constructs, 8

global construct, 8

global constructs, 8

global data, 8, 11

global dummy argument, 27

global-view model, 8

gmove, 11

image, 11

image index, 11

in mode (of gmove), 11

Intrinsic and Library Procedures

Intrinsic and Library Procedures

xmp_all_node_num, 58

xmp_all_num_nodes, 58

xmp_array_gtol, 112

xmp_desc_of, 57

xmp_malloc, 24, 58

xmp_node_num, 58

xmp_num_nodes, 58

xmp_test_async, 57

xmp_wtick, 57

xmp_wtime, 57

intrinsic transformational procedures, 111

Laplace, 137

library interface, 111

Linpack, 138

local, 8

local actual argument, 27

local alias, 11

local data, 8, 11

local dummy argument, 27

local section, 11

local-view model, 8

location-variable, 66

lock, 66

loop, 31, 46

node, 8

node array, 9

node number, 9

node reference, 17

node set, 9

nodes, 11

non-primary node array, 9

out mode (of gmove), 11

parent node set, 9

post, 64, 65

posting node, 64

primary node array, 9

primary node set, 9, 9

procedure, 8

procedure interface, 13

reduction, 18

reduction variable, 18

reflect, 39

reflection source, 25

replicate, 23

replicated data, 10

replicated execution, 8

Sample Program

Laplace, 137

Linpack, 138
sequence association, 143
shadow, 143
shadow, 144
shadow object, 144
source node, 144
structured block, 144
synchronization, 144
Syntax
  align, 22
  array, 38
  array assignment in XMP/C, 71
  array section in XMP/C, 71
  barrier, 43
  bcast, 47
  coarray, 59
  coarray reference, 60
  directive, 13
  distribute, 20
  gmove, 41
  local alias, 61
  lock, 66
  loop, 31
  node reference, 17
  nodes, 16
  post, 64
  reduction, 13
  reflect, 54
  shadow, 25
  task, 27
  tasks, 29
  template, 8
  template reference, 18
  template_fix, 20
  wait, 64
  wait_async, 64

  task, 5, 10
  task, 28, 29
  tasks, 28, 29
  template, 8
  template, 8
  template reference, 18

  unlock, 66

  variable, 109

  wait, 64, 109
  wait_async, 64
  waiting node, 52
  work mapping, 5

work mapping constructs, 5

XcalableMP C, 7
XcalableMP Fortran, 7
xmp_align_axis, 80
xmp_align_offset, 90
xmp_align_replicated, 90
xmp_align_template, 97
xmp_all_node_num, 100
xmp_all_num_nodes, 100
xmp_array_gtol, 101
xmp_array_laddr, 101
xmp_array_lead_dim, 101
xmp_array_lshadow, 101
xmp_array_lsize, 101
xmp_array_ndims, 101
xmp_array_ubound, 101
xmp_array_ushadow, 101
xmp_desc_of, 102
xmp_desc_of, 102
xmp_dist_axis, 102
xmp_dist_blocks, 102
xmp_dist_gblockmap, 102
xmp_dist_format, 102
xmp_dist_nodes, 102
xmp_finalize_mpi, 102
xmp_gather, 102
xmp_init_mpi, 102
xmp_malloc, 72
xmp_matmul, 102
xmp_node_num, 102
xmp_nodes_size, 102
xmp_nodes_attr, 102
xmp_nodes_equiv, 102
xmp_nodes_index, 102
xmp_nodes_ndims, 102
xmp_num_nodes, 102
xmp_pack, 102
xmp_scatter, 102
xmp_sort_down, 102
xmp_sort_up, 102
xmp_template_fixed, 102
xmp_template_lbound, 102
xmp_template_nndims, 102
xmp_template_ubound, 102
xmp_test_async, 102
xmpTranspose, 102
xmp_unpack, 102
xmp_wtick, X
xmp_wtime, X