# XcalableMP

⟨*ex-scalable-em-p*⟩

Language Specification

Version 1.0

XcalableMP Specification Working Group

November, 2011

Copyright ©2008-2011 XcalableMP Specification Working Group. Permission to copy without fee all or part of this material is granted, provided the XcalableMP Specification Working Group copyright notice and the title of this document appear. Notice is given that copying is by permission of XcalableMP Specification Working Group.

# Contents

**1 Introduction** **1**
1.1 Features of XcalableMP . . . . . 1
1.2 Scope . . . . . 2
1.3 Organization of this Document . . . . . 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 . . . . . 9
2.8.4 Work Terminology . . . . . 10
2.8.5 Communication and Synchronization Terminology . . . . . 10
2.8.6 Local-view Terminology . . . . . 10

**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
3.2.2 Correspondence between Node Arrays . . . . . 18
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 . . . . . . . . . . . 39
3.5.1 `reflect` Construct . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.5.2 `gmove` Construct . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.5.3 `barrier` Construct . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.5.4 `reduction` Construct . . . . . . . . . . . . . . . . . . . . . . . . . . 43
3.5.5 `bcast` Construct . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.5.6 `wait_async` Construct . . . . . . . . . . . . . . . . . . . . . . . . . 47
3.5.7 `async` Clause . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48

**4 Support for the Local-view Programming 49**
4.1 Coarrays in XcalableMP . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.1.1 `[C]` Declaration of Coarrays . . . . . . . . . . . . . . . . . . . . . . 49
4.1.2 `[C]` Reference of Coarrays . . . . . . . . . . . . . . . . . . . . . . . 50
4.1.3 `[C] sync_memory` Directive . . . . . . . . . . . . . . . . . . . . . . 50
4.2 Directives for the Local-view Programming . . . . . . . . . . . . . . . . . 51
4.2.1 `[F] local_alias` Directive . . . . . . . . . . . . . . . . . . . . . . . 51
4.2.2 `post` Construct . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.2.3 `wait` Construct . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

**5 Base Language Extensions in XcalableMP C 57**
5.1 Array Section Notation . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.2 Array Assignment Statement . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.3 Pointer to Global Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.3.1 Name of Global Array . . . . . . . . . . . . . . . . . . . . . . . . . . 59
5.3.2 The Address-of Operator . . . . . . . . . . . . . . . . . . . . . . . . 59
5.4 Dynamic Allocation of Global Data . . . . . . . . . . . . . . . . . . . . . 59
5.5 The Descriptor-of Operator . . . . . . . . . . . . . . . . . . . . . . . . . . 60

**6 Procedure Interfaces 61**
6.1 General Rule . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
6.2 Argument Passing Mechanism in XcalableMP Fortran . . . . . . . . . . . . . 61
6.2.1 Sequence Association of Global Data . . . . . . . . . . . . . . . . . . 62
6.2.2 Descriptor Association of Global Data . . . . . . . . . . . . . . . . . 65
6.3 Argument Passing Mechanism in XcalableMP C . . . . . . . . . . . . . . . . 68

**7 Intrinsic and Library Procedures 73**
7.1 System Inquiry Procedures . . . . . . . . . . . . . . . . . . . . . . . . . . 73
7.1.1 `xmp_desc_of` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
7.1.2 `xmp_all_node_num` . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
7.1.3 `xmp_all_num_nodes` . . . . . . . . . . . . . . . . . . . . . . . . . . 74
7.1.4 `xmp_node_num` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
7.1.5 `xmp_num_nodes` . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
7.1.6 `xmp_wtime` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
7.1.7 `xmp_wtick` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
7.2 Synchronization Procedures . . . . . . . . . . . . . . . . . . . . . . . . . 75
7.2.1 `xmp_test_async` . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
7.3 Miscellaneous Procedures . . . . . . . . . . . . . . . . . . . . . . . . . . 76
7.3.1 `xmp_gtol` . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
7.3.2 `[C] xmp_malloc` . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76

**Bibliography 77**

**A Programming Interface for MPI** **79**
A.1 xmp_get_mpi_comm . . . 79
A.2 xmp_init_mpi . . . 79
A.3 xmp_finalize_mpi . . . 80

**B Directive for Thread Parallelism** **81**
B.1 threads clause . . . 81

**C Interface to Numerical Libraries** **83**
C.1 Design of the Interface . . . 83
C.2 Query routines . . . 83
C.2.1 xmp_node_index . . . 84
C.2.2 xmp_node_size . . . 84
C.2.3 xmp_gt_size . . . 84
C.2.4 xmp_lt_size . . . 85
C.2.5 xmp_ga_size . . . 85
C.2.6 xmp_la_size . . . 85
C.2.7 xmp_ga_template_unitsize . . . 86
C.2.8 xmp_ga_first_idx_node_index . . . 86
C.2.9 xmp_la_lead_dim . . . 87
C.3 Example . . . 87

**D XcalableMP I/O** **91**
D.1 Categorization of I/O . . . 91
D.1.1 Local I/O . . . 91
D.1.2 Master I/O[F] . . . 91
D.1.3 Global I/O . . . 91
D.2 File Connection . . . 92
D.2.1 File Connection in Local I/O . . . 93
D.2.2 [F] File Connection in Master I/O . . . 93
D.2.3 File Connection in Global I/O . . . 93
D.3 Master I/O . . . 93
D.3.1 master_io Construct . . . 94
D.4 [F] Global I/O . . . 95
D.4.1 Global I/O File Operation . . . 96
D.4.2 Collective Global I/O Statement . . . 98
D.4.3 Atomic Global I/O Statement . . . 98
D.4.4 Direct Global I/O Statement . . . 99
D.5 [C] Global I/O Library . . . 99
D.5.1 Global I/O File Operation . . . 102
D.5.2 Collective Global I/O Functions . . . 104
D.5.3 Atomic Global I/O Functions . . . 106
D.5.4 Direct Global I/O Functions . . . 107

**E Sample Programs** **109**

# List of Figures


2.1 Hardware Model . . . . . . . . . . 3
2.2 Parallelization by the Global-view Programming Model . . . . . . . . . . 5
2.3 Local-view Programming Model . . . . . . . . . . 6
2.4 Global View and Local View . . . . . . . . . . 7

3.1 Example of Shadow of a Two-dimensional Array . . . . . . . . . . 26
3.2 Example of Periodic Shadow Reflection . . . . . . . . . . 41

6.1 Sequence Association with a Global Dummy Argument . . . . . . . . . . 63
6.2 Sequence Association with a Local Dummy Argument . . . . . . . . . . 64
6.3 Sequence Association of a Section of a Global Data as an Actual Argument with a Local Dummy Argument . . . . . . . . . . 65
6.4 Sequence Association of an Element of a Global Data as an Actual Argument with a Local Dummy Argument . . . . . . . . . . 66
6.5 Sequence Association with a Global Dummy Argument that Has Full Shadow . . 66
6.6 Descriptor Association with a Global Dummy Argument . . . . . . . . . . 68
6.7 Descriptor Association with a Local Dummy Argument . . . . . . . . . . 69
6.8 Passing to a Global Dummy Argument . . . . . . . . . . 70
6.9 Passing to a Local Dummy Argument . . . . . . . . . . 71
6.10 Passing an Element of a Global Data as an Actual Argument to a Local Dummy Argument . . . . . . . . . . 71

C.1 Invocation of a Library Routine through an Interface Procedure . . . . . . . . . . 83

# 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
- Yasuharu Hayashi ........ NEC
- Atsushi Hori ........ RIKEN
- Kohichiro Hotta ........ Fujitsu
- Hidetoshi Iwashita ........ Fujitsu
- Jinpil Lee ........ University of Tsukuba
- Yuichi Matsuo ........ JAXA
- Kazuo Minami ........ RIKEN
- Hitoshi Murai ........ RIKEN
- Kengo Nakajima ........ University of Tokyo
- Takashi Nakamura ........ JAXA
- Tomotake Nakamura ........ RIKEN
- Masahiro Nakao ........ University of Tsukuba
- Takeshi Nanri ........ Kyusyu University
- Kiyoshi Negishi ........ Hitachi
- Yasuo Okabe ........ Kyoto University
- Hitoshi Sakagami ........ NIFS
- Shoich Sakon ........ NEC
- Mitsuhisa Sato ........ University of Tsukuba
- Takenori Shimosaka ........ RIKEN
- Yoshihisa Shizawa ........ RIST
- Hitoshi Uehara ........ JAMSTEC
- Masahiro Yasugi ........ Kyoto University
- Mitsuo Yokokawa ........ RIKEN

This work is 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.

# 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.

- Extention 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 [4], Fujitsu XPF (VPP FORTRAN) [5, 6], 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

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

- Appendix A: Programming Interface for MPI
- Appendix B: Directive for Thread Parallelism
- Appendix C: Interface to Numerical Libraries
- Appendix D: XcalableMP I/O

# 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.

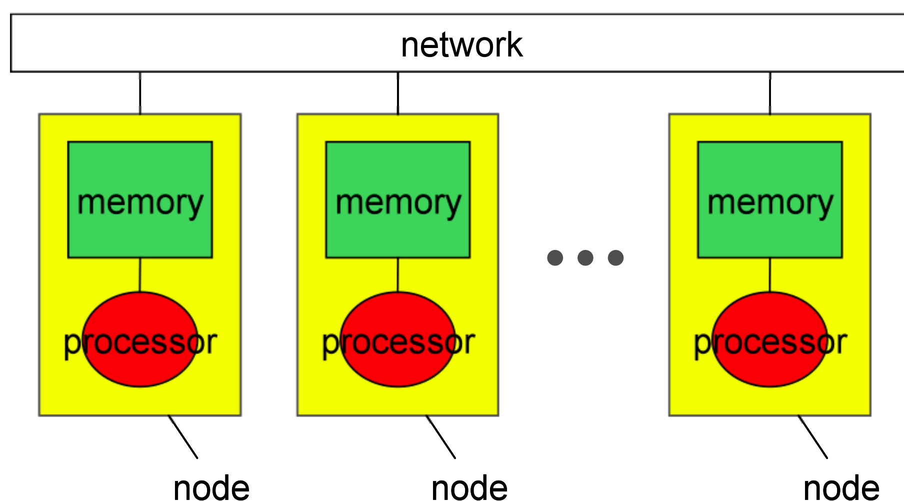


Figure 2.1: Hardware Model

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

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

The initial "current executing node set" (or the *entire node set*) at the beginning of the program execution is the set of all 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 `loop`, `array`, or `task` construct, and is contained by the node set specified by the `on` clause of the directive, it updates the current executing node set with the specified one and executes the body of the construct, after which it resumes the last executing node set and proceeds to execute the following statements.

Particularly when a node in the current executing node set encounters a `loop` or an `array` construct, it executes the loop or the array assignment in parallel with other nodes, so that each iteration of the loop or element of the assignment is independently executed by the node 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 *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: *global data* and *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 `align` directive (see section 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 `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.

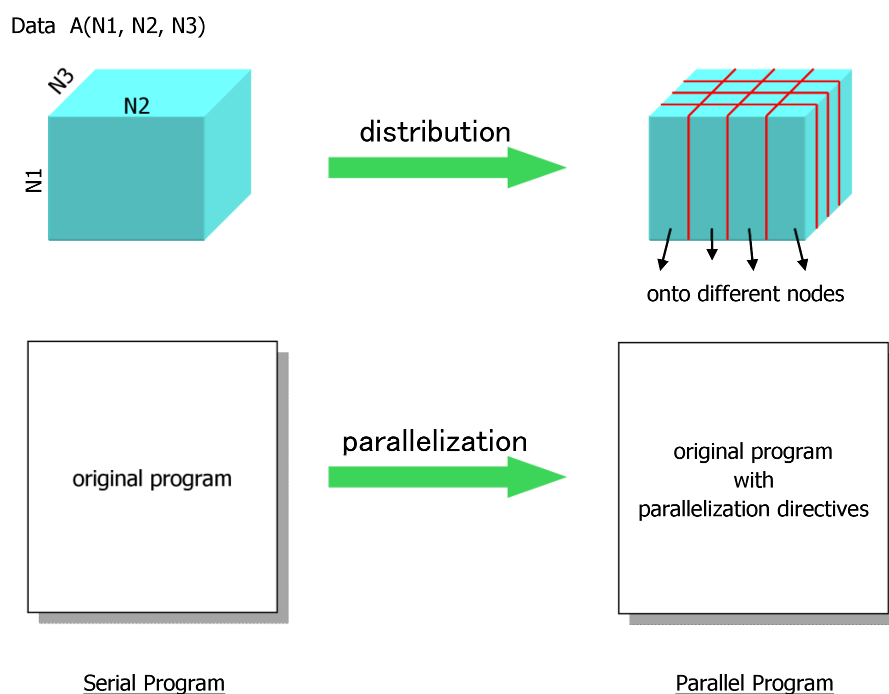


Figure 2.2: Parallelization by the Global-view Programming Model

## 2.5 Local-view Programming Model

The local-view programming model is suitable for programs that explicitly describe an algorithm and 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.

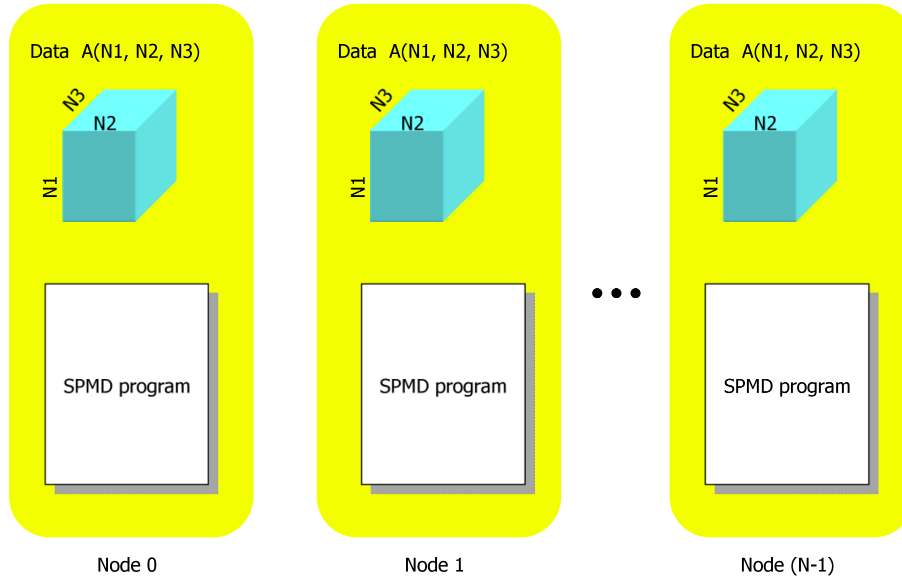


Figure 2.3: Local-view Programming Model

## 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.

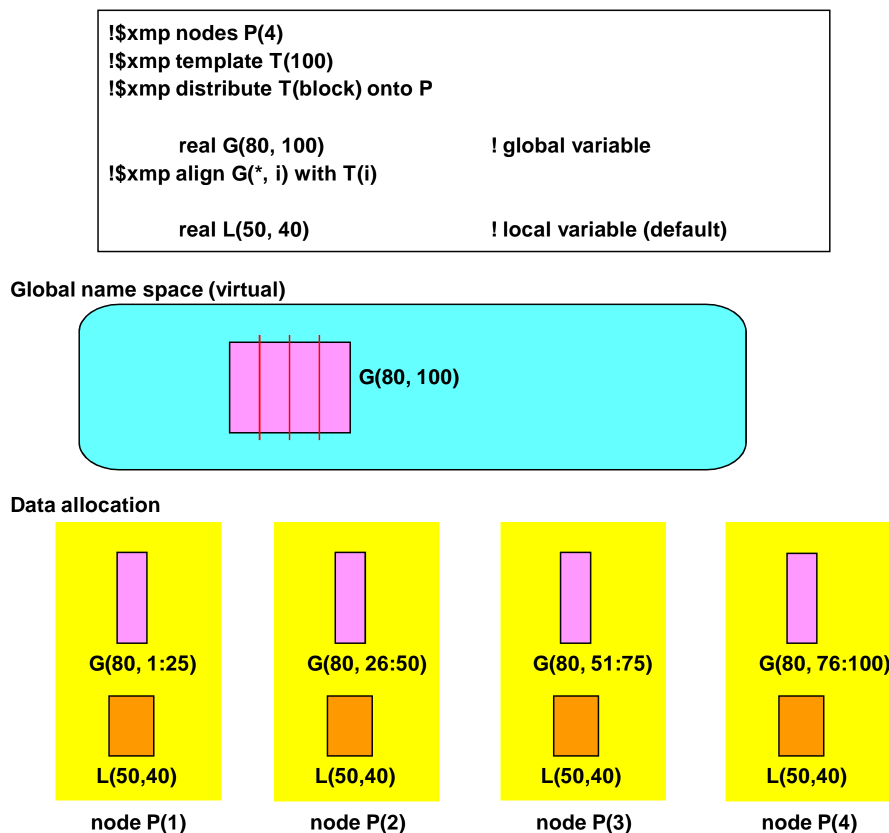


Figure 2.4: Global View and Local View

## 2.8 Glossary

### 2.8.1 Language Terminology

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

**base program** A program written in a base language.

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

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

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

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

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

**declarative directive** An XcalableMP directive that may only be placed in a declarative context. A declarative directive has no associated executable user code, 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 set of nodes.

**entire node set** A node set that contains all of the nodes participating in the execution of an XcalbleMP program. Nodes in the entire node set are linearly ordered.

**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 entire node set.

**current executing node set** An executing node set of the current context, which is managed by the XcalableMP runtimes 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.

**parent node set** The parent node set of a node set is the last executing node set, which encounterd 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 linearly-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.

001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054
055
056
057

**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 elemtent 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.

**coarray** A special local data that can be accessed directly by other nodes with a specific notation (i.e. the *image index* corresponding to the target node in the square brackets) added to the end of the array reference syntax. Every coarray is associated explicitly or implicitly with a node array and allocated on each node of the node array.

The coarray feature of XcalableMP is based on that of the Fortran 2008 standard.

**image index** A special identifier, in the style of an integer sequence, which is used in a coarray reference to specify a target node. An image index in the square brackets of a coarray reference refers to a corresponding node in the node array with which the coarray is associated. The correspondence is determined according to Fortran's array element order (i.e. row-major). Note that the lower bound of each integer (called *cosubscript*) in an image index is one in both XcalableMP Fortran and XcalableMP C.

001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054
055
056
057

# 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. |
| ▮ | The syntax rule continues. |
| `[F]` | 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.

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 *combined-directive* **is** *combined-attribute [, combined-attribute ]...* :: *combined-decl [, combined-decl ]...*

*combined-attribute* is one of:

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

and *combined-decl* is one of:

*nodes-decl*
*template-decl*
*array-name*

**Description**

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

**Restrictions**

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

## 3.2 `nodes` Directive

**Synopsis**

The `nodes` directive declares a named node array.

### Syntax

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

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

where *nodes-decl* is one of:

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

and *nodes-spec* must be one of:

*int-expr*
`*`

### Description

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

The first 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 that corresponds to the executing node set. The third form is used to declare a node array that corresponds to the node set specified by *nodes-ref*.

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.

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

XcalableMP C
```
int main() {
#pragma xmp nodes p(16)
#pragma xmp nodes q(4,*)
#pragma xmp nodes r(8)=p(3:10)
#pragma xmp nodes z(2,3)=(1:6)
    ...
}
```

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.

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

### 3.2.1 Node Reference

#### Synopsis

The node reference expression is used to reference a subset of a node set.

#### Syntax

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

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

where *nodes-subscript* must be one of:

*int-expr*
*triplet*
*

#### Description

Node reference by node number represents a node set specified by a node number of the entire node set or a triplet describing a set of node numbers of the entire node set.

Node reference by name represents a node set specified by the name of a node array or its subarray.

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 distribute directive,

  `!$xmp distribute a(block) onto p`

- To specify a node set to which the declared node array corresponds in the second form of the `nodes` directive,

```
!$xmp nodes r(2,2,4) = p(1:4,1:4)
!$xmp nodes r(2,2,4) = (1:16)
```

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

```
!$xmp task on p(1:4,1:4)
!$xmp task on (1:16)
!$xmp task on p(:,*)
!$xmp task on (m)
```

- To specify a node array with which iterations of a loop are aligned in the `loop` directive,

```
!$xmp loop (i) on p(lb(i):lb(i+1)-1)
```

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

```
!$xmp barrier on p(5:8)
!$xmp reduction (+:a) on p(*,:)
```

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

```
!$xmp bcast (b) from p(k) on p(:)
```

### 3.2.2 Correspondence between Node Arrays

If one node array and the other have the same shape and correspond to the same node set, an element of the one and an element of the other are assigned to the same node; otherwise, correspondence between any two node arrays is not specified.

## 3.3 Template and Data Mapping Directives

### 3.3.1 `template` Directive

#### Synopsis

The `template` directive declares a template.

#### Syntax

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

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

where *template-decl* is:

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

and *template-spec* must be one of:

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

**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**

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

where *template-subscript* must be one of:

*int-expr*
*triplet*
`*`

**Description**

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

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

```
!$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.

```
!$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.

```
!$xmp barrier on t(1:n)
!$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

[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)`.

`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

XcalableMP 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.

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

001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054
055
056
057

**Example 2**

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

**Example 3**

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

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:

*scalar-int-variable*
`*`
`:`

and *align-subscript* must be one of:

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

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

### Description

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

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

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

    - The second form of *align-source* is said to "collapse" the corresponding dimension of the array. As a result, the index along the corresponding dimension makes no difference in determining the alignment.
    - The second form of *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 *align-source* and the matching *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 *align-source*'s and *align-subscript*'s is determined as follows:

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

### Restrictions

- `[C]` *array-name* must be declared by a declaration statement that lexically precedes the directive.

- An align dummy variable may appear at most once in the sequence of *align-subscript*'s.

- An *align-subscript* may contain at most one occurrence of an align dummy variable.

001 • The *int-expr* in an *align-subscript* may not contain any occurrence of an align dummy
002 variable.
003
004 • The sequence of *align-sources*'s must contain exactly as many colons as the sequence of
005 *align-subscript*'s contains.
006
007 • `[F]` The array specified by *array-name* must not appear as an *equivalence-object* in an
008 `equivalence` statement.
009
010
011 • `[C]` An `align` directive for an array must precede any its appearance in the executable
012 code in the block.
013
014
015 **Examples**
016
017 **Example 1**
018
019 XcalableMP Fortran
```
!$xmp align a(i) with t(i)
```
020
021 The array element `a(i)` is aligned with the template element `t(i)`. This is equivalent to
022 the following code.
023
024 XcalableMP Fortran
025
```
!$xmp align a(:) with t(:)
```
026
027
028 **Example 2**
029
030 XcalableMP Fortran
```
!$xmp align a(*,j) with t(j)
```
031
032
033 The subarray `a(:,j)` is aligned with the template element `t(j)`. Note that the first
034 dimension of `a` is collapsed.
035
036 **Example 3**
037
038 XcalableMP Fortran
```
!$xmp align a(j) with t(*,j)
```
039
040
041 The array element `a(j)` is replicated and aligned with each template element of `t(:,j)`.
042
043 **Example 4**
044
045 XcalableMP Fortran
046
047
```
!$xmp template t(n1,n2)
      real a(m1,m2)
!$xmp align a(*,j) with t(*,j)
```
048
049
050 The subarray `a(:,j)` is aligned with each template element of `t(:,j)`.
051
052 By replacing "`*`" in the first dimension of the array `a` and "`*`" in the first dimension of the
053 template `t` with a dummy variable `i` and `k`, respectively, this alignment can be interpreted
054 as the following mapping.
055
056
057
$$a(i,j) \to t(k,j) \mid (i,j,k) \in (1:n1,\ 1:n2,\ 1:m1)$$

### 3.3.5 shadow Directive

#### Synopsis

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

#### Syntax

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

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

where *shadow-width* must be one of:

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

#### Description

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

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

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

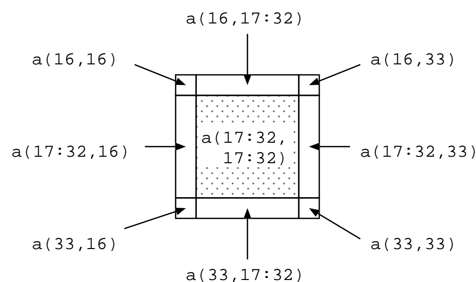


Figure 3.1: Example of Shadow of a Two-dimensional Array

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-neigboring" 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**

[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* )

**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 5.4), 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

XcalableMP Fortran

```
!$xmp template :: t(:)
!$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.

### 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(:,j)` or `a[j][:]`.

XcalableMP Fortran

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

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

      i = ...
      j = ...

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

XcalableMP C

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

    int i,j;
    float a[100][200];
#pragma align a[j][*] with t(j+1)

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

#pragma xmp task on t(j+1)
    a[j][i] = 1.0;
}
```

### 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.

XcalableMP Fortran

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

XcalableMP Fortran

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

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

**Example 2** The first node `p(1)` executes the first and 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.

XcalableMP Fortran

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

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

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

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

```
!$xmp end tasks
```

### 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

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

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

where *reduction-clause* is:

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

*reduction-kind* is one of:

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

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

and *reduction-spec* is:

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

### Description

A `loop` directive is associated with a loop nest consisting of one or more tightly-nested loops that follow the directive and 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 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 the sequence of *loop-index*'s is executed by a node specified by the

*nodes-ref*.

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

- ":" in the *nodes-ref* or the *template-ref* with "*", and
- *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
```
!$xmp loop (j) on t(:,j)
!$xmp+             reduction(op:s)
      do j = js, je
        ...
        do i = 1, N
          s = s op a(i,j)
        end do
        ...
      end do
```

XcalableMP Fortran
```
// Initialize s_tmp to the identity
// element of the op operator
      s_tmp = ...

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

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

      s = s op s_tmp
```

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.

001 • A control variable of a loop can appear as *loop-index* at most once.
002
003 • The node set specified by *nodes-ref* or *template-ref* in the `on` clause must be a subset of
004 the parent node set.
005
006 • The template specified by *template-ref* must be fixed before the `loop` construct is executed.
007
008 • The `loop` construct is global, which means that it must be executed by all of the executing
009 nodes, and each local variable referenced in the directive must have the same value among
010 all of them, and the lower bound, upper bound and step of the associated loop must have
011 the same value among all of them.
012
013 • *reduction-spec* must have one or more *location-variable*'s if and only if *reduction-kind* is
014 either `FIRSTMAX`, `FIRSTMIN`, `LASTMAX`, or `LASTMIN`.
015
016
017 **Examples**
018
019 **Example 1**

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

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

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

**Example 2**

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

```
            b(i,j) = a(i,j)
         end do
      end do
```

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.

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

**Example 3**

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

The distribution of loops in the nested loop can be specified using the sequence of *loop-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.

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

**Example 4**

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

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

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

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

**Example 5**

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

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

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

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

**Example 6**

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

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

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

**Example 7**

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

```
        end do
    end do
```

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

XcalableMP Fortran

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

This example is equivalent to the code shown below.

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

**Example 2**

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

This example is equivalent to the code shown below.

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

## 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]` `!$xmp reflect (` *array-name [, array-name]...* `)` ▮
▮ *[*`width (` *reflect-width [, reflect-width]...* `)`*]* *[*`async (` *async-id* `)`*]*

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

where *reflect-width* must be one of:

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

### Description

The `reflect` construct updates each of the shadow 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

XcalableMP Fortran

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

      real a(100)
!$xmp align a(i) with t(i)
!$xmp shdow 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)`.

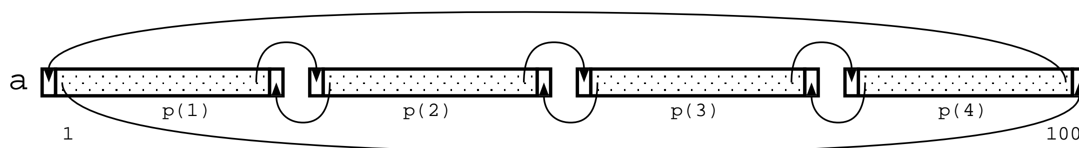


Figure 3.2: Example of Periodic Shadow Reflection

### 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 3.5.6).

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

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

**Example 2: Scalar assignment to an array** When the 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
```
!$xmp gmove
      a(:,1:N) = c(k)
```

XcalableMP C
```
#pragma xmp gmove
      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.

XcalableMP Fortran

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

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

### 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.

### 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

```
!$xmp reduction(+:s)
!$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

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

### Example 2

XcalableMP Fortran

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

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

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

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

In these cases, the reduction operation on the scalar variable `sum` is performed for every iteration in the outer loop, which may cause a large overhead. 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:

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

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

### 3.5.5 `bcast` Construct

#### Synopsis

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

### Syntax

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

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

### Description

The values of the variables specified by the sequence of *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

XcalableMP Fortran

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

The `reflect` construct on the first line matches the `wait` construct on the third line because both of their *async_id* evaluate to 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

XcalableMP adopts coarray features for the local-view programming. Particularly in XcalableMP Fortran, the features are compatible with that of Fortran 2008.

## 4.1 Coarrays in XcalableMP

The specification of the coarray features in XcalableMP conforms to that of Fortran 2008 unless otherwise provided. Each node in the entire node set is considered to correspond to each image of a program. Therefore the number of images is always equal to the size of the entire node set. The image index of a node is its node number of the entire node set.

Declaring coarrays on an arbitrary node array, which may not correspond to the entire node set, is an open issue of XcalableMP and under discussion. Note that even if the coarray features of XcalableMP would be extended in the future version so that they could be declared on a subset of the entire node set, they are compatible with those of Fortran 2008 as long as declared on the entire node set.

Terms related to coarrays in XcalableMP (e.g. coshape, coindex, cobound, cosubscript, image, image index, etc.) are derived from that in Fortran 2008.

Described in the rest of this section is the coarray features for XcalableMP C.

### 4.1.1 `[C]` Declaration of Coarrays

#### Synopsis

Coarrays are declared with the `coarray` directive in XcalableMP C.

#### Syntax

`[C]` `#pragma xmp coarray` *variable-name [*`,` *variable-name ]...* `:` *codimensions*

where *codimensions* is:

*[*`[`*int-expr*`]`*...]*`[*]`

#### Description

For XcalableMP C, coarrays are declared with the `coarray` directive where *codimensions* specify the coshape of a variable.

**Restrictions**

- A coarray specified by *variable-name* must have a global scope.

**Examples**

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

A variable `x` that has a global scope is declared as a coarray by the `coarray` directive.

### 4.1.2 [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 determine 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`.

XcalableMP C
```
int A[10], B[10];
#pragma xmp coarray A, B : [*]

A[:] = B[:]:[10];
```

### 4.1.3 [C] sync_memory Directive

**Synopsis**

In XcalableMP C, the `sync_memory` directive is used to complete all memory operations on coarrays.

**Syntax**

```
[C]  #pragma xmp sync_memory
```

### Description

The `sync_memory` directive ensures that any changes of coarrays on the image (or the executing node) done by itself are visible to any images (or nodes) and any coarrays on the image can be referenced or defined by any images. Note that no other XcalableMP directive ensures that.

The `sync_memory` directive in XcalableMP C has a function equivalent to that of the `sync memory` statement in XcalableMP Fortran.

### Examples

XcalableMP C

```
#pragma xmp coarray a, b, c, d : [*]

a = ...;
... = b;
c:[p] = ...;
... = d:[q];

#pragma xmp sync_memory
```

When a node finishes executing the `sync_memory` directive in this code snippet at runtime, the following are ensured that:

- the local definition of a coarray `a` is complete and it can be referenced and defined by any other nodes;
- the local reference of a coarray `b` is complete and it can be defined by any other nodes;
- the remote definition of a coarray `c` from the node is complete and it can be referenced and defined by any nodes; and
- the remote reference of a coarray `d` from the nodes is complete and it can be defined by any nodes.

## 4.2 Directives for the Local-view Programming

### 4.2.1 [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 distributed array.

### Syntax

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

### Description

This directive declares that a local array specified by *local-array-name* is a "local alias" to the global array specified by *global-array-name*.

The shape of a local alias is the same as that of the local section of the global array that is owned by each node.

A local alias is defined when the corresponding global array is defined. If the corresponding global array is statically allocated, then the local alias is always defined in its scoping unit; if not, the local alias is not defined until the corresponding global array is allocated.

An array specified by *local-array-name* may be a coarray.

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 can be declared if it is specified as a "local alias" by the `local_alias` directive.

In XcalableMP C, the address-of operator applied to global data substitutes for the `local_alias` directive (see 5.3).

### Restriction

- The array specified by *local-array-name* must not be aligned by an `align` directive.
- The array specified by *global-array-name* must be aligned by an `align` directive.
- The data type and rank of the array specified by *local-array-name* must be the same as those of the array specified by *global-array-name*.
- The array specified by *local-array-name* must be a deferred-shape array, which means that it must be declared with a *deferred-shape-spec-list* in a type declaration statement or a `DIMENSION` statement.

### Examples

### Example 1

XcalableMP Fortran

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

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

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

The array `a` is distributed by block onto four nodes. The node `n(2)` has its local section of 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)`.

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

**Example 2**

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

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

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

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

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

**Example 3**

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

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

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

      c

!$xmp template_fix :: t(128)

      allocate (a(128))

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

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.

001 002 003 004 005 006 007 008 009 010 011 012 013 014 015 016 017 018 019 020 021 022 023 024 025 026 027 028 029 030 031 032 033 034 035 036 037 038 039 040 041 042 043 044 045 046 047 048 049 050 051 052 053 054 055 056 057

### 4.2.2 `post` Construct

#### Synopsis

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

#### Syntax

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

#### Description

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

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

#### Example

#### Example 1

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

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

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

The `post` construct on the left matches the `wait` construct on the right because their *nodes-ref*'s represent each other and both *tags*'s have the same value of 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.

### 4.2.3 `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.

001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054
055
056
057

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

where *triplet* must be one of:

*base* : *length* : *step*
*base* : *length*
:

### 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 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 *step* is omitted, it is assumed to be "1". When all of *base*, *length* and *step* is omitted, it is assumed that *base* is "0", *length* is the size of the dimension of the array, and *step* is "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,

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

some array sections can be specified as follows:

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

## 5.2 Array Assignment Statement

### Synopsis

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

### Syntax

[C] *array-section [*: [*int-expr*] *...]* = { *variable [*: [*int-expr*] *...]* | *int-expr* };

### Description

When the rhs is an array section, the value of each element of it is assigned to the corresponding element of the lhs array section. When the rhs is an integer expression, its value is assigned to each element of the lhs array section.

The rhs and/or the lhs data can have cosubscripts.

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

### Restrictions

- [C] When the rhs is an array section, the lhs and the rhs 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.
- [C] If *variable* on the rhs is followed by ": [*int-expr*] ...", it must be a coarray.

**Examples**

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

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

## 5.3 Pointer to Global Data

### 5.3.1 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.

### 5.3.2 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.

## 5.4 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 7.3.2.

**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.

XcalableMP C
```
#pragma nodes p(NP)
#pragma xmp template t(:)
```

```
#pragma xmp distribute t(block) onto p

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);
```

## 5.5 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.

**6.1.0.0.1 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.

**6.2.1.0.2 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).

XcalableMP Fortran

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

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

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

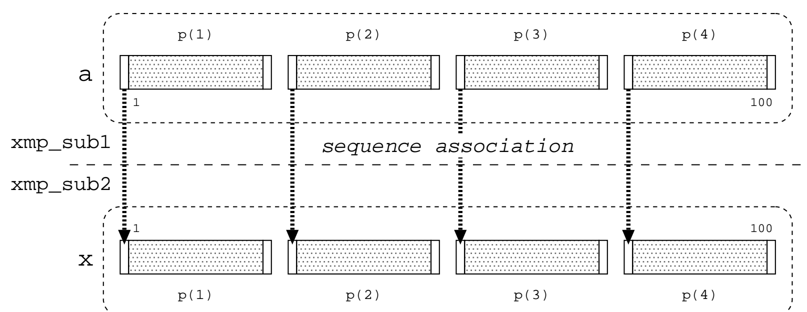


Figure 6.1: Sequence Association with a Global Dummy Argument

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

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

XcalableMP Fortran

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

Fortran

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

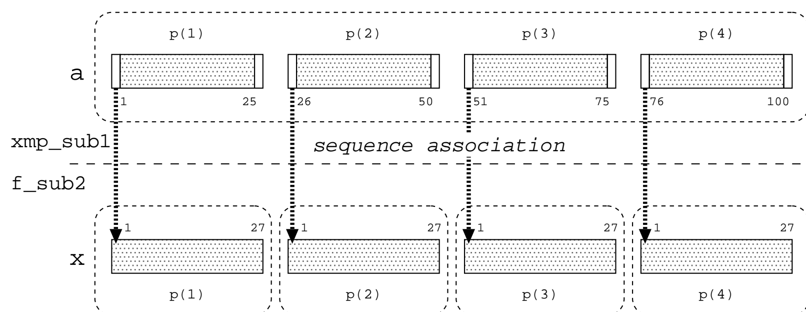


Figure 6.2: Sequence Association with a Local Dummy Argument

**Example 3** The actual argument `a(:,1)` is a contiguous section of 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_sub2` is invoked only by `p(1)` (Figure 6.3).

XcalableMP Fortran

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

Fortran

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

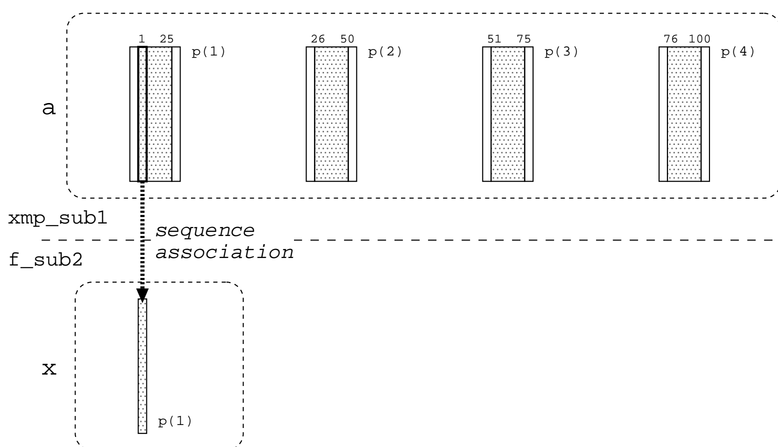


Figure 6.3: Sequence Association of a Section of a Global Data as an Actual Argument with a Local Dummy Argument

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

XcalableMP Fortran

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

Fortran

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

**Example 5** Even if either the global actual or dummy argument has a full shadow, the 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 6.5).

### 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.

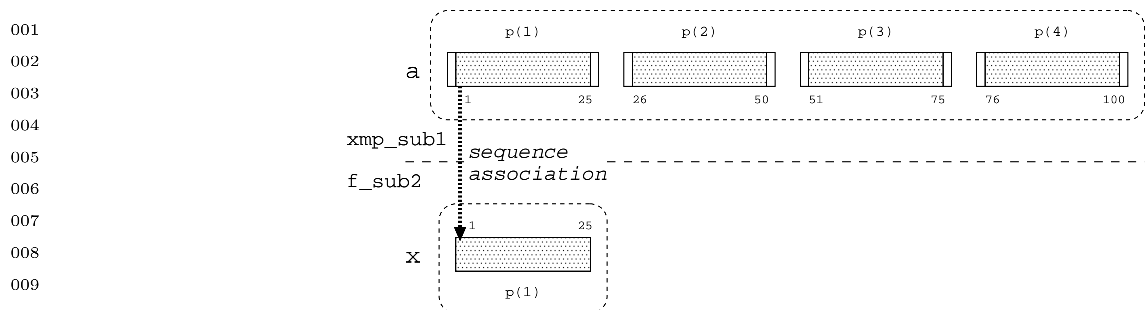


Figure 6.4: Sequence Association of an Element of a Global Data as an Actual Argument with a Local Dummy Argument

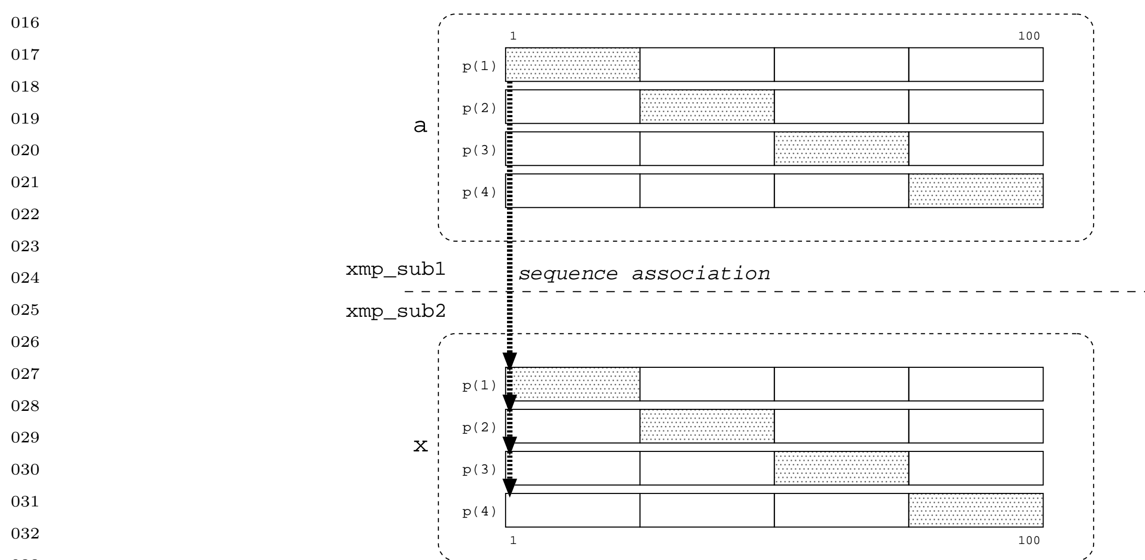


Figure 6.5: Sequence Association with a Global Dummy Argument that Has Full Shadow

**6.2.2.0.3 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 7.1.1), 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).

XcalableMP Fortran

```fortran
      subroutine xmp_sub1

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

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

      call xmp_sub2(a)

      end subroutine

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

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

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

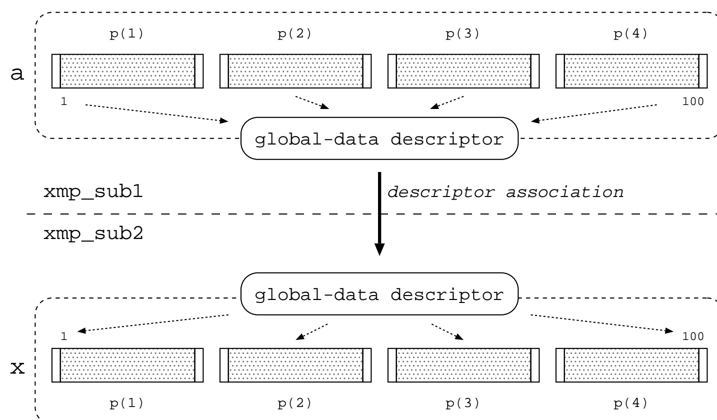


Figure 6.6: Descriptor Association with a Global Dummy Argument

XcalableMP Fortran

```
      subroutine xmp_sub1

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

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

      call f_sub2(a)

      end subroutine
```

Fortran

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

## 6.3 Argument Passing Mechanism in XcalableMP C

When an actual argument is a global data, 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.

**6.3.0.0.4 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

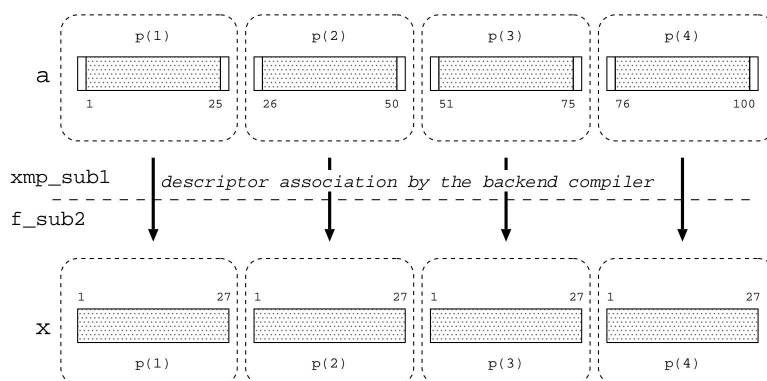


Figure 6.7: Descriptor Association with a Local Dummy Argument

each node. 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).

XcalableMP 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
```

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

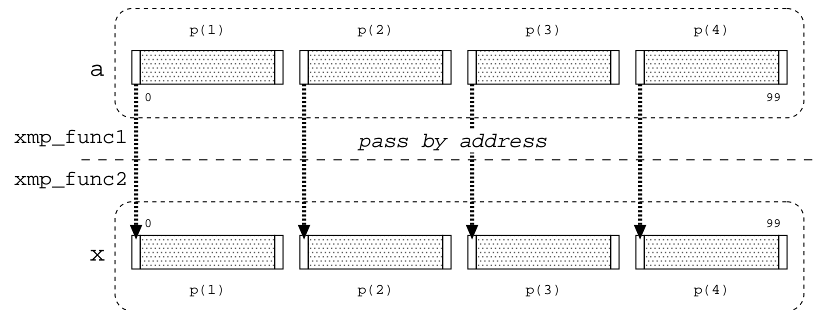


Figure 6.8: Passing to a Global Dummy Argument

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

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

XcalableMP C

```
void xmp_func1()
{
#pragma xmp nodes p(4)
#pragma xmp template t(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]

  c_func2(a);
}
```

C

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

**Example 3** The actual argument `a[0]` is an element of the global data and the dummy argument `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 `p(1)` owns the element. Hence, `c_func2` is invoked only by `p(1)` (Figure 6.10).

XcalableMP C

```
void xmp_func1()
{
#pragma xmp nodes p(4)
#pragma xmp template t(0:99)
#pragma xmp distribute t(block) onto p
```

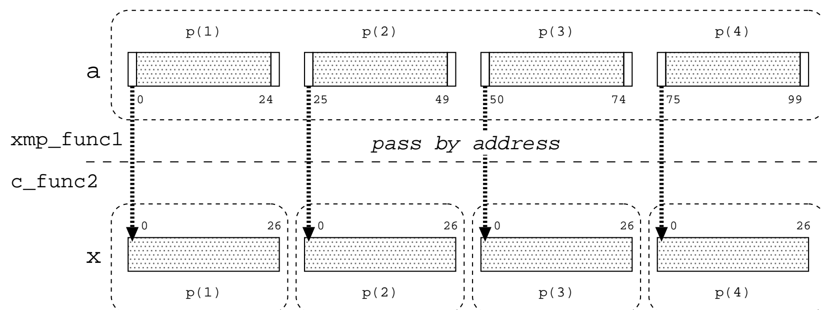


Figure 6.9: Passing to a Local Dummy Argument

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

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

C

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

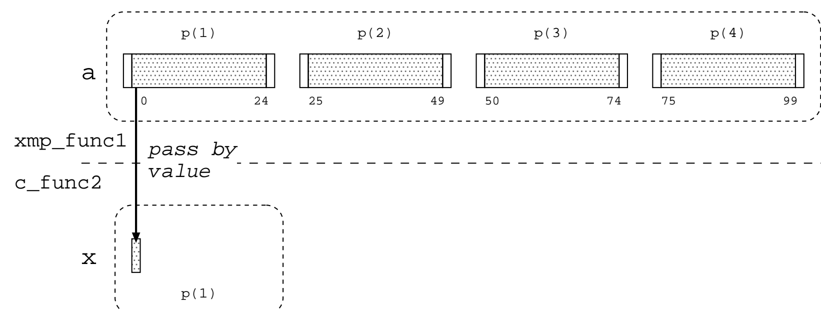


Figure 6.10: Passing an Element of a Global Data as an Actual Argument to a Local Dummy Argument

001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054
055
056
057

# 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 System Inquiry Procedures

- `xmp_desc_of`
- `xmp_all_node_num`
- `xmp_all_num_nodes`
- `xmp_node_num`
- `xmp_num_nodes`
- `xmp_wtime`
- `xmp_wtick`

### 7.1.1 `xmp_desc_of`

**Format**

```
[F]  integer(kind=xmp_desc_kind)  xmp_desc_of(xmp_entity)
[C]  xmp_desc_t                   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.

**Synopsis**

`xmp_desc_of` returns, in XcalableMP Fortran, or is evaluated to, in XcalableMP C, a descriptor to retrieve informations of the specified global array, template, or node array. The resulting descriptor can be used as an input argument of the inquiry procedures which is described in appendix C.

The kind type parameter of the type of the descriptor, `xmp_desc_kind`, in XcalableMP Fortran is implementation-dependent, and defined in a Fortran module named `xmp_lib` or a Fortran `include` file named `xmp_lib.h`.

The type of the descriptor, `xmp_desc_t`, in XcalableMP C is implementation-dependent, and defined in a header file named `xmp.h` in XcalableMP C.

#### Arguments

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

### 7.1.2 xmp_all_node_num

#### Format

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

#### Synopsis

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

#### Arguments

none.

### 7.1.3 xmp_all_num_nodes

#### Format

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

#### Synopsis

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

#### Arguments

none.

### 7.1.4 xmp_node_num

#### Format

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

#### Synopsis

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

#### Arguments

none.

### 7.1.5 xmp_num_nodes

#### Format

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

**Synopsis**

The `xmp_num_nodes` routine returns the number of the executing nodes.

**Arguments**

none.

### 7.1.6 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.

### 7.1.7 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.

## 7.2 Synchronization Procedures

### 7.2.1 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.

## 7.3 Miscellaneous Procedures

### 7.3.1 xmp_gtol

```
[F]  subroutine                     xmp_gtol(d, g_idx, l_idx)
     integer(kind=xmp_desc_kind)    d
     integer                        g_idx(NDIMS)
     integer                        l_idx(NDIMS)

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

**Synopsis**

The `xmp_gtol` routine 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 routine, the resulting array is set to an unspecified value.

**Arguments**

- `d` is a descriptor, that is, an object of type `integer(kind=xmp_desc_kind)`, in XcalableMP, or `xmp_desc_t`, in XcalableMP C, that is associated with the target 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`.
- `[F] l_idx` is a rank-one integer array of the size equal to the rank of the target global array specified by `d`.
- `[C] g_idx` is a one-dimensional integer array.
- `[C] l_idx` is a one-dimensional integer array.

### 7.3.2 [C] xmp_malloc

```
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 a descriptor specified by `d`, and returns the pointer to it on each node.

**Arguments**

- `d` is a descriptor, that is, an object of type `xmp_desc_t` that is associated with a pointer to the one-dimensional global array to be allocated.
- `size` is the size of the global array to be allocated.

# Bibliography


[1] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.1", http://www.openmp.org/mp-documents/OpenMP3.1.pdf (2011).

[2] High Performance Fortran Forum, "High Performance Fortran Language Specification Version 2.0", http://hpff.rice.edu/versions/hpf2/hpf-v20.pdf (1997).

[3] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 2.2", http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf (2009).

[4] Japan Association of High Performance Fortran, "HPF/JA Language Specification", http://www.hpfpc.org/jahpf/spec/hpfja-v10-eng.pdf (1999).

[5] Yuanyuan Zhang, Hidetoshi Iwashita, Kuninori Ishii, Masanori Kaneko, Tomotake Nakamura, and Kohichiro Hotta, "Hybrid Parallel Programming on SMP Clusters Using XPFortran and OpenMP", Proceedings of the International Workshop on OpenMP (IWOMP 2010), Vol. 6132 of Lecture Notes in Computer Science, pp. 133–148, Springer (2010).

[6] Hidetoshi Iwashita, Naoki Sueyasu, Sachio Kamiya, and Matthijs van Waveren, "VPP Fortran and the design of HPF/JA extensions", Concurrency and Computation — Practice & Experience, Vol. 14, No. 8–9, pp. 575–588, Wiley (2002).

[7] Jinpil Lee, Mitsuhisa Sato, and Taisuke Boku, "OpenMPD: A Directive-Based Data Parallel Language Extension for Distributed Memory Systems", Proceedings of the 2008 International Conference on Parallel Processing, pp. 121-128 (2008).

001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054
055
056
057

# 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

XcalableMP C

```
#include <stdio.h>
#include "mpi.h"
#include "xmp.h"

#pragma xmp nodes p(4)

int main(int argc, char *argv[]) {
  xmp_init_mpi(&argc, &argv)

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

#pragma xmp task on p(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

# Directive for Thread Parallelism

Thread-level parallelism is needed to program multi-core cluster system. Users can use some features introduced from OpenMP to parallelize loops in thread level with the `threads` clause of the `loop` directive. No direct use of OpenMP directives in XcalableMP code is allowed.

## B.1 `threads` clause

### Syntax

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

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

where *threads-clause* is:

`threads` *[ omp-clause ]*

and *omp-clause* is one of:

`num_threads(` *num-thread* `)`
`private(` *list* `)`
`firstprivate(` *list* `)`
`lastprivate(` *list* `)`

### Description

OpenMP clauses such as `num_threads` can be specified in `threads` clause. The XcalableMP compiler generates internally OpenMP directives from the `loop` directive and the `threads` clause. Note that no `reduction` need to be specified in the `threads` clause because it is inherited from the `reduction` clause in the `loop` directive.

### Example

This example calculates the total sum of an array. A `threads` clause is given to the `loop` directive to parallelize the loop statement in both process and thread level. The `reduction`

clause in the `loop` directive is also applied to the OpenMP directive which is generated by the XcalableMP compiler.

XcalableMP C

```
#include <stdio.h>
#include "xmp.h"
#define N 1024

#pragma xmp nodes p(*)
#pragma xmp template t(0:N-1)
#pragma xmp distribute t(block) onto p
#pragma xmp align a[i] with t(i)

int main(void) {
  . . . // initialize a[]

  int sum = 0;
#pragma xmp loop on t(i) reduction(+:sum) threads num_threads(4)
  for (int i = 0; i < N; i++) {
    sum += a[i];
  }

  return 0;
}
```

# Appendix C

# 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.

## C.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 C.1).

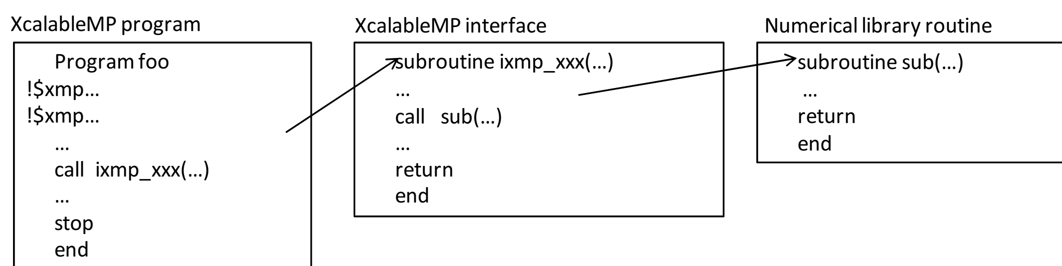


Figure C.1: Invocation of a Library Routine through an Interface Procedure

- When the numerical library routine needs information on an 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.

## C.2 Query routines

Specifications of some query routines are shown below.

### C.2.1 xmp_node_index

#### Format

```
[F]  subroutine                     xmp_node_index(d, idx)
     integer(kind=xmp_desc_kind)    d
     integer                        idx(dim)
[C]  void                           xmp_node_index(xmp_desc_t d, int idx[])
```

#### Synopsis

The `xmp_node_index` routine provides the indices of the executing node in the target node array.

#### Arguments

- `d` is a descriptor, that is, an object of type `integer(kind=xmp_desc_kind)`, in XcalableMP, or `xmp_desc_t`, in XcalableMP C, that is associated with the node array.
- `idx` is a one-dimensional integer array. `dim` is the rank of the node array.

### C.2.2 xmp_node_size

#### Format

```
[F]  subroutine                     xmp_node_size(d, size)
     integer(kind=xmp_desc_kind)    d
     integer                        size(dim)
[C]  void                           xmp_node_size(xmp_desc_t d, int size[])
```

#### Synopsis

The `xmp_node_size` routine provides the size of each dimension of the target node array.

#### Arguments

- `d` is a descriptor, that is, an object of type `integer(kind=xmp_desc_kind)`, in XcalableMP, or `xmp_desc_t`, in XcalableMP C, that is associated with the node array.
- `size` is a one-dimensional integer array. `dim` is the rank of the node array.

### C.2.3 xmp_gt_size

#### Format

```
[F]  subroutine                     xmp_gt_size(d, size)
     integer(kind=xmp_desc_kind)    d
     integer                        size(dim)
[C]  void                           xmp_gt_size(xmp_desc_t d, int size[])
```

#### Synopsis

The `xmp_gt_size` routine provides the global size of each dimension of the target template.

### Arguments

- `d` is a descriptor, that is, an object of type `integer(kind=xmp_desc_kind)`, in XcalableMP, or `xmp_desc_t`, in XcalableMP C, that is associated with the target template.
- `size` is a one-dimensional integer array. `dim` is the rank of the template.

## C.2.4 xmp_lt_size

### Format

```
[F]  subroutine                      xmp_lt_size(d, size)
     integer(kind=xmp_desc_kind)     d
     integer                         size(dim)
[C]  void                            xmp_lt_size(xmp_desc_t d, int size[])
```

### Synopsis

The `xmp_lt_size` routine provides the local size of each dimension of the target template.

### Arguments

- `d` is a descriptor, that is, an object of type `integer(kind=xmp_desc_kind)`, in XcalableMP, or `xmp_desc_t`, in XcalableMP C, that is associated with the template.
- `size` is a one-dimensional integer array. `dim` is the rank of the template.

## C.2.5 xmp_ga_size

### Format

```
[F]  subroutine                      xmp_ga_size(d, size)
     integer(kind=xmp_desc_kind)     d
     integer                         size(dim)
[C]  void                            xmp_ga_size(xmp_desc_t d, int size[])
```

### Synopsis

The `xmp_ga_size` routine provides the global size of each dimension of the target global array.

### Arguments

- `d` is a descriptor, that is, an object of type `integer(kind=xmp_desc_kind)`, in XcalableMP, or `xmp_desc_t`, in XcalableMP C, that is associated with the global array.
- `size` is to be set to a one-dimensional integer array. `dim` is the rank of the global array.

## C.2.6 xmp_la_size

### Format

```
[F]  subroutine                      xmp_la_size(d, size)
     integer(kind=xmp_desc_kind)     d
     integer                         size(dim)
[C]  void                            xmp_la_size(xmp_desc_t d, int size[])
```

001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054
055
056
057

#### Synopsis

The `xmp_la_size` routine provides the local size of each dimension of the global array.

#### Arguments

- `d` is a descriptor, that is, an object of type `integer(kind=xmp_desc_kind)`, in XcalableMP, or `xmp_desc_t`, in XcalableMP C, that is associated with the global array.
- `size` is a one-dimensional integer array. `dim` is the rank of the global array.

### C.2.7 xmp_ga_template_unitsize

#### Format

```
[F]  subroutine                   xmp_ga_template_unitsize(d, unitsize)
     integer(kind=xmp_desc_kind)  d
     integer                      unitsize(dim)
[C]  void                         xmp_ga_template_unitsize(xmp_desc_t d, int unitsize[])
```

#### Synopsis

The `xmp_ga_template_unitsize` routine provides the blocking factor of each dimension of the target template.

#### Arguments

- `d` is a descriptor, that is, an object of type `integer(kind=xmp_desc_kind)`, in XcalableMP, or `xmp_desc_t`, in XcalableMP C, that is associated with the template.
- `unitsize` is a one-dimensional integer array. `dim` is the rank of the template.

### C.2.8 xmp_ga_first_idx_node_index

#### Format

```
[F]  subroutine                   xmp_ga_first_idx_node_index(d, idx)
     integer(kind=xmp_desc_kind)  d
     integer                      idx(dim)
[C]  void                         xmp_ga_first_idx_node_index(xmp_desc_t d, int idx[])
```

#### Synopsis

The `xmp_ga_first_idx_node_index` routine provides the indices of the node onto which the *first* element of the global array is distributed.

#### Arguments

- `d` is a descriptor, that is, an object of type `integer(kind=xmp_desc_kind)`, in XcalableMP, or `xmp_desc_t`, in XcalableMP C, that is associated with the global array.
- `idx` is a one-dimensional integer array. `dim` is the rank of node array associated with the global array.

### C.2.9 xmp_la_lead_dim

**Format**

| | | |
|---|---|---|
| [F] | subroutine | xmp_la_lead_dim(d, lead_dim) |
| | integer(kind=xmp_desc_kind) | d |
| | integer | lead_dim |
| [C] | void | xmp_la_lead_dim(xmp_desc_t d, int lead_dim) |

**Synopsis**

The xmp_la_lead_dim routine provides the leading dimension of each local section of the target global array.

**Arguments**

- d is a descriptor, that is, an object of type integer(kind=xmp_desc_kind), in XcalableMP, or xmp_desc_t, in XcalableMP C, that is associated with the global array.
- lead_dim is an integer scalar.

## C.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 Subprograms). ScaLAPACK library routines invoked from XcalableMP procedures also depend on BLACS.

**Example 1** This example shows an implementation of the interface for the ScaLAPACK driver routine pdgesv.

XcalableMP Fortran

```
      subroutine ixmp_pdgesv(n,nrhs,a,ia,ja,da,ipiv,b,ib,jb,db,ictxt,info)

      use xmp_lib

      integer n,nrhs,ia,ja,ib,jb,ictxt,info
      double precision a,b
      integer(kind=xmp_desc_kind) da,db
      integer size_a(2),unitsize_a(2),rank_a(2),lead_dim_a,desca(9)
      integer size_b(2),unitsize_b(2),rank_b(2),lead_dim_b,descb(9)

      call xmp_ga_size(da,size_a)
      call xmp_ga_template_unitsize(da,unitsize_a)
      call xmp_ga_first_idx_nodes_rank(da,rank_a)
      call xmp_la_lead_dim(da,lead_dim_a)

      call xmp_ga_size(db,size_b)
      call xmp_ga_template_unitsize(db,unitsize_b)
      call xmp_ga_first_idx_nodes_rank(db,rank_b)
      call xmp_la_lead_dim(db,lead_dim_b)
```

```
      desca(1)=1
      desca(2)=ictxt
      desca(3)=size_a(1)
      desca(4)=size_a(2)
      desca(5)=unitsize_a(1)
      desca(6)=unitsize_b(2)
      desca(7)=rank_a(1)
      desca(8)=rank_a(2)
      desca(9)=lead_dim_a

      descb(1)=1
      descb(2)=ictxt
      descb(3)=size_b(1)
      descb(4)=size_b(2)
      descb(5)=unitsize_b(1)
      descb(6)=unitsize_b(2)
      descb(7)=rank_b(1)
      descb(8)=rank_b(2)
      descb(9)=lead_dim_b

      call pdgesv(n,nhrs,a,ia,ja,desca,ipiv,b,ib,jb,descb,info)

      return
      end
```

**Example 2** This example shows an XcalableMP procedure using the interface of Example 1.

XcalableMP Fortran

```
      program xmptdgesv

      use xmp_lib

      double precision a(1000,1000)
      double precision b(1000)
      integer ipiv(2*1000,2)
!$xmp nodes p(2,2)
!$xmp template t(1000,1000)
!$xmp template t1(2*1000,2)
!$xmp distribute t(block,block) onto p
!$xmp distribute t1(block,block) onto p
!$xmp align a(i,j) with t(i,j)
!$xmp align ipiv(i,j) with t1(i,j)
!$xmp align b(i) with t(i,*)
      ...
      integer i,j,ictxt
      integer m=1000,n=1000,nprow=2,npcol=2
      integer icontxt=-1,iwhat=0
      integer nrhs=1,ia=1,ja=1,ib=1,jb=1,info
      character*1 order
```

```
      ...
      order="C"
      ...
25    call blacs_get(icontxt,iwhat,ictxt)
      call blacs_gridinit(ictxt,order,nprow,npcol)
      ...
!$xmp loop (i,j) on t(i,j)
      do j=1,n
30       do i=1,m
            a(i,j) = ...
         end do
      end do
      ...
35 !$xmp loop on t(i,*)
      do i=1,m
         b(i)= ...
      end do
      ...
40    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)
      ...
45    stop
      end
```

001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054
055
056
057

# 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[F]

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 local. In a large system with many

Table D.1: Global I/O

| | independent/collective | access method |
|---|---|---|
| Collective I/O | collective | sequential access |
| Atomic I/O | independent | sequential access |
| Direct I/O | independent | direct access |

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 include the 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 global (collective) execution and sequential file access. It handles global data in the sequential order, similar to master I/O. **Atomic** global I/O is local 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 local 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. File system visible to each node is implementation dependent.

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 same file by multiple nodes behave; e.g. , data in the standard input file may be read only by one node or may be replicated to all nodes. It is implementation dependent how data from the multiple nodes are merged into the standard output/error file.

### 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 global (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 substring-range
  - A section-subscript
  - An expression including operators
  - An *io-implied-do-control*
- An I/O statement specified with a master I/O directive must be executed collectively on the node set that is connected to the file.
- Internal file I/O is not 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 global (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

| | I/O item | operation |
|---|---|---|
| input item | name of global array | The data that is read from the file in the sequential order of array elements is distributed onto the global array on the node set. The file positioning increases by the size of data. |
| | array element of global array | The data 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. |
| | local variable | The data 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. |
| | implied DO loop | For each input item, repeat the above operation. |
| output item | name of global array | The value of the global array is collected and is written to the file in the sequential order of array elements. The file positioning increases by the size of data. |
| | array element of global array | The value of the element of the global array is written to the file. A file position increases by the size of data. |
| | local variable and expression | The value evaluated on the master node is written to the file. The file positioning increases by the size of data. |
| | implied DO loop | For each output item, repeat the above operation. |

Namelist input and output statements cannot treat global arrays. A namelist output statement writes the values on the master node to the file. In the namelist input, each item of the namelist is read from the file to the master node if it is recorded in the file. 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.

001 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:

| I/O statement | available specifiers |
|---|---|
| OPEN | UNIT, IOSTAT, FILE, STATUS, POSITION, ACTION, FORM |
| CLOSE | UNIT, IOSTAT, STATUS |
| READ | UNIT, IOSTAT |
| WRITE | UNIT, IOSTAT |

Case of `global_io` construct with a direct clause:

| I/O statement | available specifiers |
|---|---|
| OPEN | UNIT, IOSTAT, FILE, STATUS, RECL, ACTION, FORM |
| CLOSE | UNIT, IOSTAT, STATUS |
| READ | UNIT, REC, IOSTAT |
| WRITE | UNIT, REC, IOSTAT |

An input item and an output item of a data transfer statement with `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 `global_io` directive.

The OPEN statement specified with a `global_io` directive is global (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 global (collective) execution and must be executed by the same executing node set as the one where the OPEN statement is executed.

Utilizable values of the specifiers in I/O statements are shown in the following table. Definitions of the specifiers are described in the specification of the base language.

- OPEN statement

| specifiers | value | default |
|---|---|---|
| UNIT | external file unit (scalar constant expression) | not omissible |
| FILE | file name (scalar CHARACTER expression) | not omissible |
| STATUS | 'OLD', 'NEW', 'REPLACE' or 'UNKNOWN' | 'UNKNOWN' |
| POSITION | 'ASIS', 'REWIND' or 'APPEND' | 'ASIS' |
| ACTION | 'READ', 'WRITE' or 'READWRITE' | processor dependent |
| RECL | the value of the record length (scalar constant expression) | not omissible |
| FORM | 'FORMATTED' or 'UNFORMATTED' | The default value is FORMATTED if the file is begin connected with a direct clause, and the default value is UNFORMATTED if the file is begin connected without a direct clause. |

POSITION is available only if the directive has no direct clause. RECL is available only if the directive has a direct clause.

- CLOSE statement

| specifiers | value | default |
|---|---|---|
| UNIT | external file unit (scalar constant expression) | not omissible. |
| STATUS | 'KEEP' or 'DELETE' | 'KEEP' |

- READ/WRITE statement

| specifiers | value | default |
|---|---|---|
| UNIT | external file unit (scalar constant expression) | not omissible |
| REC | the value of the number of record (scalar constant expression) | not omissible |

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**

`!$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 in global (collective). 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 (in global), while atomic READ and WRITE statements are executed in local. A file connected by an atomic OPEN statement can be disconnected only by an atomic CLOSE statement executed on the same

| | I/O item | operation |
|---|---|---|
| input item | name of global array | The values read from a file are assigned to the elements of the global array. The file positioning seeks by the size of the data. |
| | local variable | The values read from the file are replicated into the local array on all executing nodes. The file positioning seeks by the length of the data. |
| output item | name of global array | The values of a global array are written to the file in the sequential order of the array elements. The file positioning seeks by the size of the data. |
| | local variable, expression | The values evaluated on a node of the executing nodes are written to the file. The file positioning seeks by the size of the data. |

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 (in global), while direct READ and WRITE statements are executed in local. 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 in local. 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.

| **function name** | **xmp_range_t *xmp_allocate_range(n_dim)** | |
|---|---|---|
| argument | int n_dim | the number of dimensions |
| return value | xmp_range_t* | descriptor of array section. NULL is returned when a program abend. |

| **function name** | **void xmp_set_range(rp, i_dim, lb, length, step)** | |
|---|---|---|
| argument | xmp_range_t *rp | descriptor |
| | int i_dim | target dimension |
| | int lb | lower bound of array section in the dimension i_dim |
| | int length | length of array section in the dimension i_dim |
| | int step | stride of array section in the dimension i_dim |

| **function name** | **void xmp_free_range(rp)** | |
|---|---|---|
| argument | xmp_range_t *rp | descriptor of array section. |

### D.5.1 Global I/O File Operation

#### D.5.1.1 xmp_fopen_all

xmp_fopen_all opens a global I/O file. Collective (global) execution.

| **function name** | **xmp_file_t *xmp_fopen_all(fname, amode)** | |
|---|---|---|
| argument | const char *fname | file name |
| | const char *amode | equivalent to fopen of POSIX. combination of “rwa+” |
| return value | xmp_file_t* | file structure. NULL is returned when a program abend. |

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

| amode | intended purpose |
|---|---|
| r | Open for reading only. File pointer points the beginning of the file. |
| r+ | Open an existing file for update (reading and writing). File pointer points the beginning of the file. |
| w | Create for writing. If a file by that name already exists, it will be overwritten. File pointer points the beginning of th file. |
| w+ | 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. |
| a | 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. |
| a+ | 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. |

#### D.5.1.2 xmp_fclose_all

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

| **function name** | **int *xmp_fclose_all(fh)** | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| return value | int | 0: normal termination<br>1: abnormal termination. fh is NULL.<br>2: abnormal termination. error in MPI_File_close. |

#### D.5.1.3 xmp_fseek

xmp_fseek sets the individual file pointer in the file structure. Local execution.

| **function name** | **int xmp_fseek(fh, offset, whence)** | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | long long offset | displacement of current file view from position of whence |
| | int whence | choose file position<br>SEEK_SET: the beginning of the file<br>SEEK_CUR: current position<br>SEEK_END: the end of the file |
| return value | int | 0: normal termination<br>an integer other than 0: abnormal termination |

#### D.5.1.4 xmp_fseek_shared

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

| **function name** | **int xmp_fseek_shared(fh, offset, whence)** | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | long long offset | displacement of current file view from position of whence |
| | int whence | choose file position<br>SEEK_SET: the beginning of the file<br>SEEK_CUR: current position<br>SEEK_END: the end of the file |
| return value | int | 0: normal termination<br>an integer other than 0: abnormal termination |

#### D.5.1.5 xmp_ftell

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

| **function name** | **long long xmp_ftell(fh)** | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| return value | long long | Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, negative number shall be returned. |

#### D.5.1.6 xmp_ftell_shared

xmp_ftell_shared inquires the position of shared file pointer in the file structure. Local execution.

| function name | long long xmp_ftell_shared(fh) | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| return value | long long | Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, negative number shall be returned. |

#### D.5.1.7 xmp_file_sync_all

xmp_file_sync_all guarantees completion of access to the file from nodes sharing the file. Two data accesses conflict if they access the same absolute byte displacements of the same file and at least one is a write access. When two accesses A1 and A2 to the same file conflict in direct or collective I/O, an xmp_file_sync_all to the file must be invoked between A1 and A2, otherwise the outcome of the accesses is undefined. Collective (global) execution.

| function name | int xmp_file_sync_all(fh) | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| return value | int | 0: normal termination<br>an integer other than 0: abnormal termination |

### D.5.2 Collective Global I/O Functions

Collective I/O is executed collectively (in global) 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 (global) execution.

| function name | int xmp_file_set_view_all(fh, disp, desc, rp) | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | long long disp | displacement from the beginning of the file. |
| | xmp_desc_t desc | descriptor |
| | xmp_range_t *rp | range descriptor |
| return value | int | 0: normal termination<br>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 (global) execution.

The positions of the shared and individual file pointers are set to disp and the elemental data type and the file type are set to MPI_BYTE.

| **function name** | **int xmp_file_clear_view_all(fh, disp)** | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | long long disp | displacement from the beginning of the file. |
| return value | int | 0: normal termination<br>an integer other than 0: abnormal termination |

#### D.5.2.3 xmp_fread_all

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

| **function name** | **size_t xmp_fread_all(fh, buffer, size, count)** | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | void *buffer | beginning address of loading variables |
| | size_t size | the size of a loading element of data |
| | size_t count | the number of loading data element |
| return value | size_t | Upon successful completion, return the size of loading data. Otherwise, negative number shall be returned. |

#### 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 (global) execution.

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

| **function name** | **size_t xmp_fwrite_all(fh, buffer, size, count)** | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | void *buffer | beginning address of storing variables |
| | size_t size | the size of a storing element of data |
| | size_t count | the number of storing data element |
| return value | size_t | Upon successful completion, return the size of storing data. Otherwise, negative number shall be returned. |

#### D.5.2.5 xmp_fread_darray_all

xmp_fread_darray_all reads data cooperatively to the global array from the position of the shared file pointer.

Data is read from the file to distributed *desc* limited to range *rp*.

| function name | size_t xmp_fread_darray_all(fh, desc, rp) | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | xmp_desc_t desc | descriptor |
| | xmp_range_t *rp | range descriptor |
| return value | size_t | Upon successful completion, return the size of loading data. Otherwise, negative number shall be returned. |

#### D.5.2.6 xmp_fwrite_darray_all

xmp_fwrite_darray_all writes data cooperatively from the global array to the position of the shared file pointer.

| function name | size_t xmp_fwrite_darray_all(fh, desc, rp) | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | xmp_desc_t desc | descriptor |
| | xmp_range_t *rp | range descriptor |
| return value | size_t | Upon successful completion, return the size of loading data. Otherwise, negative number shall be returned. |

Data is written from distributed *desc* limited to range *rp* to the file.

### D.5.3 Atomic Global I/O Functions

Atomic I/O is executed in local 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. Local execution.

| function name | size_t xmp_fread_shared(fh, buffer, size, count) | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | void *buffer | beginning address of loading variables |
| | size_t size | the size of a loading element of data |
| | size_t count | the number of loading data element |
| return value | size_t | Upon successful completion, return the size of loading data. Otherwise, negative number shall be returned. |

#### D.5.3.2 xmp_fwrite_shared

xmp_fwrite_shared exclusively writes local data to the position of the shared file pointer and moves the position by the length of the data. Local execution.

| **function name** | **size_t xmp_write_shared(fh, desc, rp)** | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | xmp_desc_t desc | descriptor |
| | xmp_range_t *rp | range descriptor |
| return value | size_t | Upon successful completion, return the size of storing data. Otherwise, negative number shall be returned. |

### D.5.4 Direct Global I/O Functions

Direct I/O is executed in local 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 a 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. Local execution.

| **function name** | **size_t xmp_fread(fh, buffer, size, count)** | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | void *buffer | beginning address of loading variables |
| | size_t size | the size of a loading element of data |
| | size_t count | the number of loading data element |
| return value | size_t | Upon successful completion, return the size of loading data. Otherwise, negative number shall be returned. |

#### D.5.4.2 xmp_fread

xmp_fread writes data to the position of the individual file pointer and moves the position by the length of the data. Local execution.

| function name | size_t xmp_fwrite(fh, buffer, size, count) | |
|---|---|---|
| argument | xmp_file_t *fh | file structure |
| | void *buffer | beginning address of storing variables |
| | size_t size | the size of a storing element of data |
| | size_t count | the number of storing data element |
| return value | size_t | Upon successful completion, return the size of storing data. Otherwise, negative number shall be returned. |

# Appendix E

# Sample Programs

**Example 1**

XcalableMP C

```
/*
 *  A parallel explicit solver of Laplace equation in \XMP
 */
#pragma xmp nodes p(NPROCS)
#pragma xmp template t(1:N)
#pragma xmp distribute t(block) onto p

double u[XSIZE+2][YSIZE+2],
       uu[XSIZE+2][YSIZE+2];
#pragma xmp align u[i][*] to t(i)
#pragma xmp align uu[i][*] to t(i)
#pragma xmp shadow uu[1:1][0:0]

lap_main()
{
 int x,y,k;
 double sum;
for(k = 0; k < NITER; k++){
        /* old <- new */
#pragma xmp loop on t(x)
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            uu[x][y] = u[x][y];
#pragma xmp reflect (uu)
#pragma xmp loop on t(x)
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            u[x][y] = (uu[x-1][y] + uu[x+1][y] +
                  uu[x][y-1] + uu[x][y+1])/4.0;
    }

    sum = 0.0;
#pragma xmp loop on t[x] reduction(+:sum)
    for(x = 1; x <= XSIZE; x++)
        for(y = 1; y <= YSIZE; y++)
```

```
          sum += (uu[x][y]-u[x][y]);
#pragma xmp task on p(1)
    printf("sum = %g\n",sum);
}
```

**Example 2**

XcalableMP C

```
/*
 * Linpack in XcalableMP (Gaussian elimination with partial pivoting)
 *    1D distribution version
 */
#pragma xmp nodes p(*)
#pragma xmp template t(0:LDA-1)
#pragma xmp distribute t(cyclic) onto p

double pvt_v[N];  // local

/*     gaussian elimination with partial pivoting        */
dgefa(double a[n][LDA],int lda, int n,int ipvt,int *info)
#pragma xmp align a[:][i] with t(i)
{
    REAL t;
    int idamax(),j,k,kp1,l,nm1,i;
    REAL x_pvt;

    nm1 = n - 1;
    for (k = 0; k < nm1; k++) {
      kp1 = k + 1;
      /* find l = pivot index        */
      l = A_idamax(k,n-k,a[k]);
      ipvt[k] = l;

      /* if (a[k][l] != ZERO) */
#ifdef XMP
#pragma xmp gmove
      pvt_v[k:n-k] = a[l][k:n-k];
#else
      for(i = k; i < n; i++) pvt_v[i] = a[i][l];
#endif

      /* interchange if necessary */
      if (l != k){
#ifdef XMP
#pragm xmp gmove
        a[l][:] = a[k][:];
#pramga xmp gmove
        a[k][:] = pvt_v[:];
#else
        for(i = k; i< n; i++) a[i][l] = a[i][k];
        for(i = k; i< n; i++) a[i][k] = pvt_v[i];
```

```
#endif
      }
      /* compute multipliers */
      t = -ONE/pvt_v[k];
      A_dscal(k+1, n-(k+1),t,a[k]);

      /* row elimination with column indexing */
      for (j = kp1; j < n; j++) {
        t = pvt_v[j];
        A_daxpy(k+1,n-(k+1),t,a[k],a[j]);
      }
    }
    ipvt[n-1] = n-1;
}

dgesl(double a[n][LDA],int lda,int n,int pvt[n],double b,int job)
#pragma xmp align a[:][i] with t(i)
#pragma xmp align b[i] with t(i)
{
    REAL t;
    int k,kb,l,nm1;

    nm1 = n - 1;
    /* job = 0 , solve  a * x = b,  first solve  l*y = b  */
    for (k = 0; k < nm1; k++) {
        l = ipvt[k];
#pragma xmp gmove
        t = b[l];
        if (l != k){
#pragma xmp gmove
            b[l] = b[k];
#pragma xmp gmove
            b[k] = t;
        }
        A_daxpy(k+1,n-(k+1),t,a[k],b);
    }

    /* now solve u*x = y */
    for (kb = 0; kb < n; kb++) {
        k = n - (kb + 1);
#pragma xmp task on t(k)
{
        b[k] = b[k]/a[k][k];
        t = -b[k];
}
#pragma xmp bcast (t) from t(k)
        A_daxpy(0,k,t,a[k],b);
    }
}
```

```
/*
* distributed array based routine
*/
A_daxpy(int b,int n,double da,double dx[n],double dy[n])
#pragma xmp align dx[i] with t(i)
#pragma xmp align dy[i] with t(i)
{
    int i,ix,iy,m,mp1;
    if(n <= 0) return;
    if(da == ZERO) return;
    /* code for both increments equal to 1 */
#pragma xmp loop on t(b+i)
    for (i = 0;i < n; i++) {
        dy[b+i] = dy[b+i] + da*dx[b+i];
    }
}

int A_idamax(int b,int n,double dx[n])
#pragma xmp align dx[i] with t(i)
{
  double dmax, g_dmax;
    int i, ix, itemp;
    if(n == 1) return(0);

    /* code for increment equal to 1 */
    itemp = 0;
    dmax = 0.0;
#pragma xmp loop on t(i) reduction(lastmax:dmax/itemp/)
    for (i = b; i < n; i++) {
      if(fabs((double)dx[i]) > dmax) {
        itemp = i;
        dmax = fabs((double)dx[i]);
      }
    }
    return (itemp);
}

A_dscal(int b,int n,double da,double dx[n])
#pragma xmp align dx[i] with t(i)
#pragma xmp align dy[i] with t(i)
{
    int i;
    if(n <= 0)return;

    /* code for increment equal to 1 */
#pragma xmp loop on t(i)
    for (i = b; i < n; i++)
      dx[i] = da*dx[i];
}
```