

XcalableMP
<ex-scalable-em-p>
Application Program Interface Version 1

DRAFT 0.7

XcalableMP Specification Working Group

November, 2009

Contents

1	Introduction	1
1.1	Scope	1
1.2	Features of XcalableMP	1
2	Overview of the XcalableMP model and language	3
2.1	Hardware model and execution model	3
2.2	Global-view programming model	4
2.3	Local-view programming model	5
2.4	Interactions between global view and local view	5
2.5	Execution model and task	6
2.6	Glossary	6
3	Directives	10
3.1	Node declaration	11
3.1.1	Nodes directive	11
3.1.2	Node reference	12
3.2	Template and data mapping	14
3.2.1	Template directive	14
3.2.2	Template reference	15
3.2.3	Distribute directive	15
3.2.4	Align directive	18
3.2.5	Shadow directive	20
3.2.6	template_fix directive	20
3.3	Work mapping construct	22
3.3.1	Task construct	22
3.3.2	Tasks construct	22
3.3.3	Loop construct	23
3.3.4	Array construct	30
3.4	Global-view communication and synchronization construct	31
3.4.1	Reflect construct	31
3.4.2	Gmove construct	32
3.4.3	Barrier construct	33
3.4.4	Reduction construct	34
3.4.5	Bcast construct	37
4	Support for local-view programming	38
4.1	Coarray notation of XcalableMP	38
4.2	Coarray in the C language	40
4.2.1	Array section notation	40
4.2.2	Assignment of array sections	40

4.2.3	Declarations and reference of coarray	41
4.3	Directive for local-view programming	41
4.3.1	<code>local_alias</code> directive	41
4.3.2	Post construct	44
4.3.3	Wait construct	45
4.3.4	Critical construct	45
5	Procedure call and data mapping for procedure argument	46
6	Runtime library	47
7	Other issues	48
7.1	Hybrid programming with MPI and OpenMP	48
7.1.1	MPI	48
7.1.2	OpenMP	48
7.2	Interface to numerical libraries	48
7.2.1	ScaLAPACK	48
8	Sample Programs	49

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

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

Chapter 1

Introduction

This document specifies a collection of compiler directives, i.e., runtime library routines that can be used to specify distributed-memory parallel programming in C and FORTRAN programs. These compiler directives define the specifications of the XcalableMP Application Program Interface (XcalableMP API). These specifications provide a model of parallel programming for distributed memory multiprocessor systems. The directives extend the C and FORTRAN base languages to describe distributed memory parallel programs.

1.1 Scope

XcalableMP API is used to explicitly specify the action to be taken by the compiler and the runtime system to execute the parallel program in a distributed memory system. XcalableMP-compliant implementations are not required to check for invalid local data access, data conflicts, racing conditions, or deadlock. The XcalableMP is defined by following items:

- A set of directives
- Minimum language extension on base languages (C and FORTRAN)
- Runtime libraries
- Environment Variables

1.2 Features of XcalableMP

The features of XcalableMP are summarized as follows:

Language extensions for familiar languages, such as C and FORTRAN, that can reduce code-rewriting and educational costs.

XcalableMP supports typical parallelization based on the **data parallel paradigm** and work sharing under *global view* and enables parallelization of the original sequential code with minimal modification using simple **directives**, such as OpenMP .

XcalableMP also includes a CAF-like Partitioned Global Address Space (PGAS) feature as *local-view* programming.

Explicit communication and synchronization. All actions are taken by directives for being “easy-to-understand” for performance-aware programming

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

For flexibility and extensibility, the execution model allows **combination with explicit MPI 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 experience obtained in the development of HPF, Fujitsu XPF (VPP FORTRAN), and OpenMPD.

Chapter 2

Overview of the XcalableMP model and language

2.1 Hardware model and execution model

The target of XcalableMP is a distributed memory system (Figure 2.1). Each computation node, which may have several cores sharing main memory, has its own local memory, and each node is connected via network. Each node can access and modify its local memory directly and can access the memory on other nodes via communication. However, it is assumed that accessing remote memory is much slower than accessing local memory.

The basic execution model of XcalableMP is a Single Program Multiple Data (SPMD) model on distributed memory. In each node, a program starts from the same main routine. An XcalableMP program begins as a single thread of execution in each node.

When the thread encounters XcalableMP directives, synchronization and communication occurs between nodes. In other words, no synchronization or communications occur without directives. In this case, the program performs duplicate execution of the same program on local memory in each node.

OpenMP API can be used in order to make use of multicores in a node. In this specification, we define actions only when XcalableMP directives are executed one thread at a time .

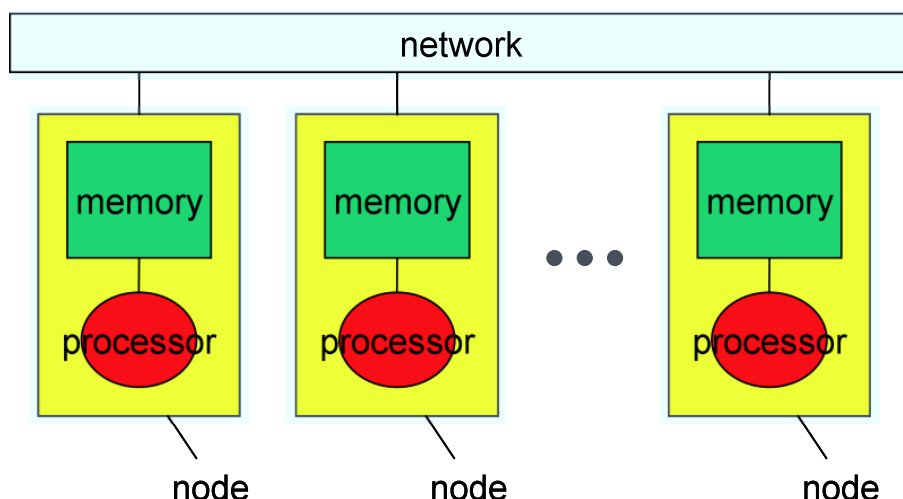


Figure 2.1: Hardware Model

001 By default, data declared in the program are allocated in each node and are referenced locally
002 by threads executed in the node.

003 XcalableMP supports two models of data viewing: the global-view programming model and
004 the local-view programming model. In the local-view programming model, accesses to data in
005 remote nodes is performed explicitly by language extension for get/put operations on remote
006 nodes with the node number of the target nodes, while reference to local data is executed
007 implicitly.
008

009 In contrast to a local-view programming model, a global-view programming model is a model
010 in which programmers express their algorithm and data structure in their entirety, mapping them
011 to the node set. The programmers describe the data distribution and the work mapping in order
012 to express how to distribute data and share the workload among nodes. The variables in the
013 global-view programming model appear as a shared memory spanning the nodes.
014

015 016 **2.2 Global-view programming model** 017

018 The global-view programming model is useful when, starting from sequential version of the
019 program, the programmer parallelizes the program in the data-parallel model by adding direc-
020 tives incrementally with minimum modification. In the global-view programming model, the
021 programmer describes the data distribution of the data shared among the nodes by data dis-
022 tribution directives. `loop` iteratively construct maps to the node where the computed data is
023 located. Global-view communication directives are used for synchronization between nodes, to
024 maintain the consistency of the shadow area, and to move part of the distributed data globally.
025 Note that the programmer must perform all computations that require data reference locally by
026 any appropriate directives.
027

028 In many cases, the XcalableMP program using the global-view programming model is based
029 on a sequential program and can produce the same results independent of the number of compu-
030 tation nodes (Figure 2.2). The global view provides a programming model in which computation
031 and data are distributed onto computation nodes.
032

033 There are three groups of directives for the global-view programming model. Since these
034 directives can be ignored as a comment by the compilers of base languages (C and FORTRAN
035), an XcalableMP program derived from a sequential program can preserve the integrity of the
036 original program when the program is run sequentially.
037

038 039 **Data Mapping**

040 Specify the data distribution and mapping to nodes (partially inherited from HPF).
041

042 043 **Work Mapping (Parallelization)**

044 Assign tasks to nodes. The `loop` construct maps each iteration to nodes containing the referenced
045 elements, and the `Task` construct executes each task in parallel in different node sets.
046

047 048 **Communication and Synchronization**

049 Describe how to communicate and synchronize with the other compute nodes. In XcalableMP ,
050 inter-node communication must be described explicitly. The compiler guarantees that commu-
051 nication takes place only if communication is explicitly specified.
052
053
054
055
056
057

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

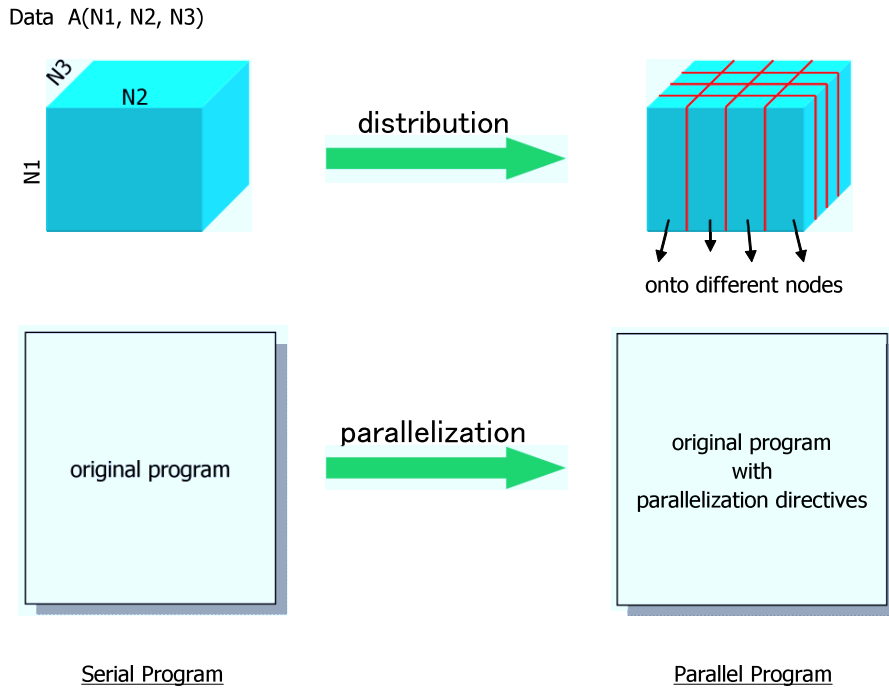


Figure 2.2: Parallelization by the global-view programming model

2.3 Local-view programming model

The local-view programming model is suitable for programs that explicitly describe the algorithm of each node and for explicit remote data reference (Figure 2.3). Since MPI is considered to have a local view, the local-view programming model of XcalableMP has high interoperability with MPI.

For the local-view programming model, the language extension and some directives are provided. The coarray notation taken from Co-array Fortran (CAF) is such an extension. For example, in order to access an array element of $A(i)$ located on compute node N , the expression of $A(i)[N]$ is used. If the access is a value reference, then communication to obtain the value occurs. If the access is to update the value, then communication to set a new value occurs.

2.4 Interactions between global view and local view

The node in XcalableMP is used to distribute data or the computational load. In the local view, the node is used in conjunction with the coarray directive to reference data. 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 a global view and local view and can be accessed from either view. XcalableMP provides some directives to set the local name (alias) to the global data declared in the global-view programming mode so that the data can also be referenced in the local-view programming model. It may be useful to optimize the program by explicit remote data reference in the local-view programming model.

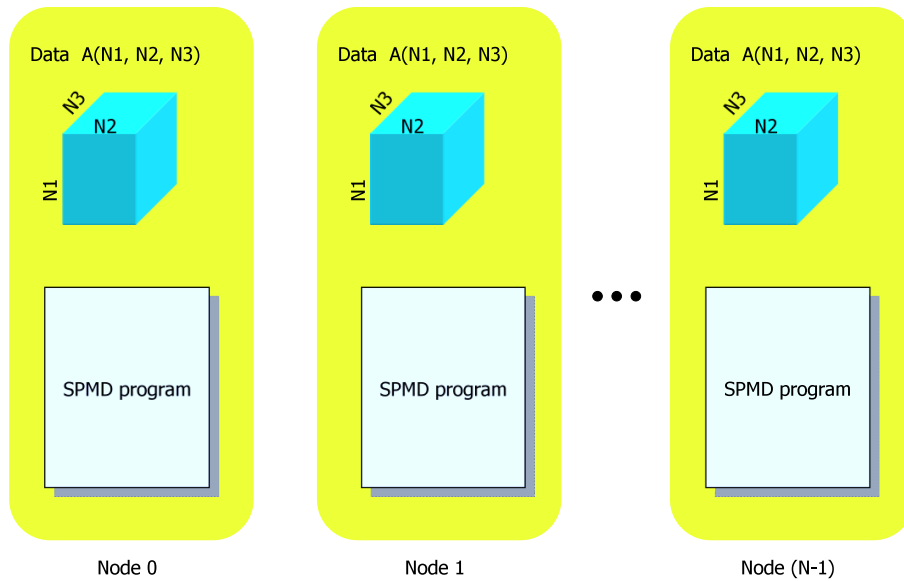


Figure 2.3: Local-view programming model

2.5 Execution model and task

In XcalableMP, a program begins as a single thread of execution in each node. The set of nodes when starting a program is referred to as the entire node set.

A task is a specific instance of executable code and its data environment executed in a set of nodes. A task when starting a program in the entire node set is called an initial task. The initial task can generate a subtask, which is executed on a subset of the nodes by the `task` construct. A set of nodes executing the same task is referred to as the set of executing nodes. If no `task` construct is encountered, then a program is executed as a single task, and its executing nodes are the entire node set.

If no directives are encountered, then a program is executed locally. When the same codes are executed, almost the same computation is performed in each node, which is referred to as duplicate execution. When the threads encounter a `loop` construct or an `array` construct, the specified loop is executed in parallel, so that each iteration is assigned to the node where the specified data element is located.

A new task is generated by the `task` construct. A code in the `task` construct is executed as a subtask executed in a specified node set. When a subroutine is called in the context of the task, the subroutine is executed on its executing nodes.

For synchronization and communication between nodes, a set of directives is provided. In the local-view programming model, coarray features are adopted for remote data reference. Note that all synchronization and communication are specified explicitly by directives, and without such directives, no communications are executed implicitly by the compiler.

2.6 Glossary

node

In a distributed memory system, a computation node, which may have several cores sharing main memory, has its own local memory. Each node is connected through a network. An XcalableMP program begins as a single thread of execution in each node.

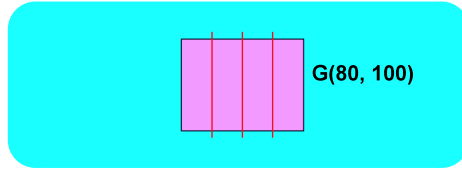
```

001 dimension L(50, 40)
002 dimension G(80, 100)
003 !$xmp template, distribute (block) onto (0:4) :: T(100)
004 !$xmp align with T :: G

```

! local view (default)
! global view

Global name space (virtual)



Data allocation

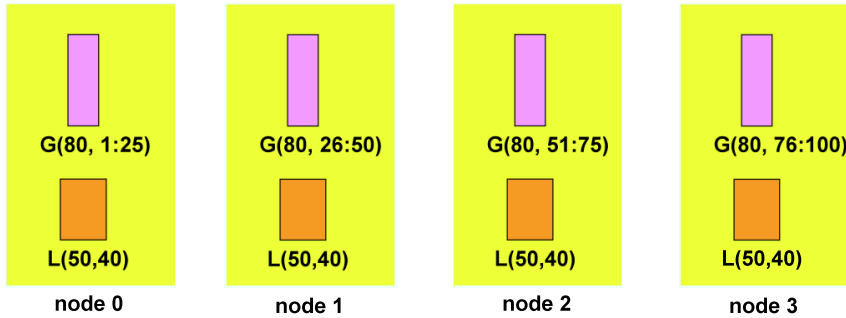


Figure 2.4: Global view and local view

node number

A unique number assigned to each of the nodes in the entire node set. The number starts from 1, larger than or equal to 1 and less than and equal to the number of nodes. Note that the mapping from the node number to the MPI rank is decided by the system. The image index of the coarray mapping to the entire node set is equal to the node number.

node set

A set of nodes

entire node set, entire set of nodes

All nodes executing the program, or a set of these nodes. The entire node set is decided when starting the program.

executing node set, executing nodes

A node set executing a certain region of a program. The executing node set that executes an entire program is the entire node set. The executing node set of a task is the node set that executes the task.

node array

A multi-dimensional array containing nodes. The node array has a name and shape as it attributes.

001 **executing node array**

002 Node array that contains the executing node.
003

004
005 **task**

006 A specific instance of executable code and its data environments executed in a set of nodes. In
007 the context of the program text, a set of statement executed by a set of nodes. A task can be
008 nested, and a nested task is executed as a subtask of an outer task.
009

010
011 **template**

012 A dummy array used to express an index space associated with an array. Template is also used
013 to describe the iteration space of a loop. A template has a name, a dimension, and an upper
014 and lower bound for each dimension as attributes.
015
016

017
018 **replicated execution**

019 Execution of the same code in different nodes. If the state at the starting point is the same and
020 the execution has only local side-effects, then the local state in each node remains the same.
021

022
023 **data mapping**

024 The combination of the alignment and distribution attributes used to describe how a data object
025 is allocated to nodes.
026

027
028 **work mapping**

029 Assignment of iterations to nodes in a parallel loop and tasks to nodes.
030

031
032 **image index**

033 A number assigned to each image of the coarray. The value is equal to or greater than 1. Note
034 that the image index of the coarray mapping to the entire node set is identical to the node
035 number.
036
037

038
039 **local**

040 Execution of a program has side-effects only on the data in the node. In this case, no commu-
041 nication with other nodes occurs .
042

043
044 **non-local**

045 Execution of a program requires communication with other nodes and has side-effects with
046 respect to other nodes.
047

048
049 **global data**

050 Data declared as a distributed array and shared by nodes.
051
052
053
054
055
056
057

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

local data

Data is allocated in each node and is referenced only within the node.* collective An operation must be executed by every node in the executing node set in order to perform an operation together.

distribution

The partition of the index space of a data object among a set of nodes according to a given pattern. The `distribute` directive is used to map the elements of a template onto a set of nodes.

alignment

An attribute of a data object that establishes the relationship between data objects for distribution. The `align` directive is used to describe the correspondence of the element of the data and the template.

shadow

A data area is used to keep neighbor elements temporarily in a distributed array. Shadow is an attribute of a distributed array that is declared by the `shadow` directive and is updated by the `reflect` directive.

Chapter 3

Directives

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

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

In C , XcalableMP directives are specified using the **#pragma** mechanism provided by C standards. In FORTRAN , XcalableMP directives are specified using special comments that are identified by unique sentinels **!\$xmp**. The rules of FORTRAN directives in fixed source format and free source format follow those in OpenMP and HPF.

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

[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 one of the declaration statements in the base language. For example, in C, node declarations by **node** directives are effective from the declared point to the end of the file, and, in FORTRAN , node declarations by **node** directives are effective within the subprogram.

Executable directives are placed in executable context. A stand-alone directive is an executable directive that has no associated user code, such as a **barrier** directive. Some executable directives compose directive constructs using an associated user code, as in the following format:

[F] **!\$xmp** *directive-name clause ...*
 statement
 ...
 !\$xmp end *directive-name*

Note that in the **loop** construct, **end** can be omitted.

[F] **!\$xmp** *directive-name clause ...*
 do-loop-construct

001 [C] #pragma xmp *directive-name clause ...*
002 *statement*

003 In C, the statement can be a compound-statement.

005 3.1 Node declaration

007 3.1.1 Nodes directive

009 Synopsis

011 The nodes directive declares a node array with a name, a shape, and some attributes.

013 Syntax

015 [F] !\$xmpnodes [(*map-type*)] *nodes-name(nodes-size)* [,*nodes-name(nodes-size)*]...
016 [F] !\$xmpnodes [(*map-type*)] *nodes-name(nodes-size)* [,*nodes-name(nodes-size)*]...=*
017 [F] !\$xmpnodes [(*map-type*)] *nodes-name(nodes-size)* [,*nodes-name(nodes-size)*]...=*nodes-ref*
018
019 [C] #pragma xmpnodes [(*map-type*)] *nodes-name(nodes-size)* [,*nodes-name(nodes-size)*]...
020 [C] #pragma xmpnodes [(*map-type*)] *nodes-name(nodes-size)* [,*nodes-name(nodes-size)*]...=*
021 [C] #pragma xmpnodes [(*map-type*)] *nodes-name(nodes-size)* [,*nodes-name(nodes-size)*]...=*nodes-ref*
022

023 where *nodes-size* is a positive integer value or *map-type* is **regular**.

025 Description

027 The node directive declares a node array to express a set of nodes in the context. The first form
028 is used to declare the entire node set. The second and third forms declare a new node array with
029 a name, a dimension, and a size in order to reference a set of nodes. The second form is used to
030 declare the executing node set. The ee*" symbol specifies the current executing node set. The
031 third form declares a node array in order to reference the node set specified by *nodes-ref*.

033 If *map-type* is specified as **regular**, then the order of nodes in the node array follows that of
034 the FORTRAN array. Therefore, in the first form, the node number is used to order nodes in
035 the node array with FORTRAN array ordering. In the second and third forms, the nodes are
036 ordered according to the sequence association with referenced nodes.

037 If no *map-type* is specified, then the ordering nodes in the node array are system dependent.
038 It is desirable to order the nodes in order to make use of the network topology for efficient
039 communication.

040 If the last *node-size* is ee*", then the size is automatically adjusted according to the total
041 size of the referenced node sets.

043 Restrictions

- 046 • *nodes-name* is an identifier in class (1) and must not conflict with other names in class (1).
- 047 • In FORTRAN, the second form cannot be used in the main program or module.
- 048 • *nodes-size* can be ee*" only in the last dimension.
- 049 • The node name referenced in *nodes-ref* must not reference *nodes-name* directly or indirectly.
- 050 • If *nodes-size* does not contain ee*", then the product of all *nodes-size* must be equal to
051 the total size of the referenced node set. The referenced node set consists of all nodes in
052 the first form, the executed node set in the second form, and the node set referenced by
053 *nodes-ref* in the third form.

- nodes-subscript in *nodes-ref* must not be *ee**”.

Examples

The following are examples of the first and third forms in the main program. Since the declaration of node array *p* specifies 16 nodes as its size, this program must be executed with 16 nodes. Since *regular* is not specified, it is not guaranteed that *Ar*(1) and *p*(3) are the same node, and the node number of *z*(1,1) is 1.

FORTRAN	C
<pre> 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 </pre>	<pre> 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) ... } </pre>

Example using the *regular* option. Since node array *p* is declared without the *regular* option, it is not guaranteed that *p*(1), *p*(2) have the node number 1, 2, ... and so on. The node array *q* with the *regular* option has the order in which *q*(1,1), *q*(2,1), *q*(3,1), *q*(4,1), *q*(1,2), ... have node numbers 1,2,3,4,5, ... In node array *z* with the *regular* option, *z*(1,1), *z*(2,1), *z*(1,2), *z*(2,2), *z*(1,3), *z*(2,3), ... have the node numbers 1, 2, 3, 4, 5, 6, ...

FORTRAN
<pre> program main !\$xmp nodes p(16) !\$xmp nodes(regular) q(4,*) !\$xmp nodes(regular) r(8)=p(3:10) !\$xmp nodes(regular) z(2,3)=(1:6) ... end program </pre>

Example using a subprogram. Since the node declaration has the second form, the caller of the *foo* subroutine must be executed with 16 nodes. The declaration for the node array *q* of the first form declares the node array for the entire node set. The node array *r* is a subset of *p*, and the node array of *x* is a subset of *q*.

FORTRAN
<pre> 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 </pre>

3.1.2 Node reference

Synopsis

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

Syntax

nodes-ref is either node reference by node number, *node-number-ref*, or node reference by name, *named-nodes-ref*.

<i>nodes-ref</i>	<i>node-number-ref</i> — <i>named-nodes-ref</i>
<i>node-number-ref</i>	<i>node-number</i> — (<i>node-number</i> [: <i>node-number</i>][: <i>int-expr</i>])
	<i>node-number</i> is a positive number.
<i>named-nodes-ref</i>	<i>nodes-name</i> [(<i>nodes-subscript</i> [, ...])]
<i>nodes-subscript</i>	<i>int-expr</i> — <i>triplet</i> — *

Description

Node reference by node number refers one node by the node number, or a node set by a sequence of node numbers.

Node reference by name refers the node set by the node array name or the subset of the node set by a subarray of the node array.

The subscript of the subarray of a node array must be either an integer, a triplet, or *ee**. The notation of the subarray using a triplet in the subscript is the same as that in FORTRAN .

The *ee** symbol in *nodes-subscript* in a subarray of a node array specifies a subscript associated with the executing node in the node array of the executing node set. Thus, the following node is referenced by name with the *k*-th subscript *ee**:

$$p(s_1, \dots, s_{k-1}, *, s_{k+1}, \dots, s_n)$$

where, with the exception of s_k , subscripts s_i must not be *ee**, is evaluated at the node

$$p(j_1, \dots, j_{k-1}, j_k, j_{k+1}, \dots, j_n)$$

where j_i is an integer, in

$$p(s_1, \dots, s_{k-1}, j_k, s_{k+1}, \dots, s_n).$$

This node reference composes the node set using nodes with the *k*-th subscript j_k . The same rule is applied even if more than two subscripts are *ee**. This notation can be used only in the node reference of the *on* clause in executable directives.

Examples

Assume that *p* is a nodes name and that *m* is an integer variable.

- Target node array by the `distribute` directive

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

- Target the subarray of the node array in the `nodes` directives

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

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

- In the `task` directive, a set of executing nodes is specified for the task.

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

```
!$xmp task on (1:16)
```

```
!$xmp task on p(:,*)
```

```
!$xmp task on m
```

- In the `loop` directive, sets of executing nodes are specified for the iterations. `!$xmp loop (i) on p(lb(`

- In barrier directive and the reduction directive, executing nodes are specified.

```
001
002     !$xmp barrier on p(5:8)
003     !$xmp reduction (+:a) on p(*,:)
004
```
- In the bcast directive, a source node and executing nodes are specified.

```
005
006     !$xmp bcast b from p(k) on p(:)
007
008
```

Examples

<pre> 010 _____ FORTRAN _____ 011 subroutine caller 012 !\$xmp nodes p(1000) 013 real a(100,100) 014 ... 015 016 5 !\$xmp tasks 017 !\$xmp task on p(1:500) 018 call task1(a) 019 !\$xmp end task 020 !\$xmp task on p(501:800) 021 call task1(a) 022 10 !\$xmp end task 023 !\$xmp task on p(801:1000) 024 call task1(a) 025 !\$xmp end task 026 !\$xmp end tasks 027 15 !\$xmp end tasks 028 ... 029 end do 030</pre>	<pre> _____ FORTRAN _____ subroutine task1(a) ... !\$xmp nodes q(*) real a(100,100) ... end subroutine 5</pre>
--	--

3.2 Template and data mapping

3.2.1 Template directive

Synopsis

The `template` directive declares a template.

Syntax

[F] `!$xmp template-name (template-spec [, template-spec] ...)`

[C] `#pragma xmp template-name (template-spec [, template-spec] ...)`

where *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*. If all expressions in the sequence of *template-spec* are `ee:"`, then the shape of the template

001 is initially undefined. This template must not be referenced until the shape is defined by *tem-*
002 *plate_fix* directives at run-time. If *int-expr* is specified as *template-spec*, then the default lower
003 bound is 1.
004

005 Restrictions

- 006 • Each *template-spec* must be either all [*int-expr* :] *int-expr* or all ee:”.

009 3.2.2 Template reference

011 Synopsis

012 The template reference expression is used to reference a subset of the referenced template.
013

015 Syntax

016
$$\begin{array}{l} \textit{template-ref} \qquad \textit{template-name} [(\textit{template-subscript} [, \dots])] \\ \textit{template-subscript} \quad \textit{int-expr} \text{ --- } \textit{triplet} \text{ --- } * \end{array}$$

020 Description

021 The template reference refers to a subarray of the template array.

022 The subscript of the subarray of a template array must be either an integer, a triplet, or ee*”.
023 The notation of the subarray using a triplet in the subscript is the same as that in FORTRAN .
024

027 Examples

028 Assume that *t* is a template name.
029

- 030 • In the `task` directive, a set of executing nodes is indirectly specified for the task.
031 `!$xmp task on t(1:m,1:n)`
032 `!$xmp task on t`
033
- 034 • In the `loop` directive, an element of the template at which each loop iteration is aligned.
035 `!$xmp loop (i) on t(i-1)`
036
- 037 • In the `array` directive, a template which the following array assignment statement is
038 aligned with.
039 `!$xmp array on t(1:n)`
040
- 041 • In the `barrier`, `reduction`, and `bcast` directives, executing nodes are specified indirectly.
042 `!$xmp barrier on t(1:n)`
043 `!$xmp reduction (+:a) on t(*,:)`
044 `!$xmp bcast b from p(k) on t(1:n)`
045
046
047
048

049 3.2.3 Distribute directive

051 Synopsis

052 The `distribute` directive specifies a distribution of templates.
053
054
055
056
057

Syntax

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

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

where *dist-format* must be one of:

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

Description

According to the specified distribution format, a template is distributed onto a set of nodes. The dimension of the node set appearing in an `onto` clause corresponds, in left-to-right order, with the dimension of the distributed template for which the corresponding *dist-format* is not `ee*`.

The interpretation of *dist-format* is as follows:

* The corresponding dimension is not distributed.

block The corresponding dimension of the template is divided into contiguous blocks of the same size, which are distributed onto the corresponding dimension of the node set. Let *d* be the size of the corresponding dimension of the template, and let *p* be the size of the corresponding dimension of node sets. If $d \bmod p$ is not zero, then the dimension of the template is divided into $d/\text{ceil}(d/p)$ blocks of size $\text{ceil}(d/p)$ and one block of size $d\% \text{ceil}(d/p)$, and each block is assigned sequentially to an index along the corresponding dimension of the node set. Note that if $k=p-d/\text{ceil}(d/p)-1 > 0$, then there is no block for the last *k* indices.

cyclic Equivalent to `cyclic(1)`.

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

gblock(m) *m* is a mapping array. The corresponding 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 set.

If more than one `gblock(*)` is specified in *dist-format*, then the template must not be referenced until the shape of the template is defined by *template_fix* directives at run-time.

Restrictions

- The number of *dist-format*, which is not `ee*`, must be equal to the rank of the node set 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 set.

- The element of the *int-array* array in parentheses following `gblock` must be a non-negative integer.
- The sum of the elements of the *int-array* array in parentheses following `gblock` must be equal to or greater than the size of the corresponding dimension of the template specified by *template-name*.

Examples

Example 1

```

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

```

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

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

Example 2

```

                                FORTRAN
program main
!$xmp nodes p(4)
!$xmp template t(64)
!$xmp distribute t(cyclic(8)) onto p

```

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

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

```

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

```

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

p(1)	t(1:64, 1:57:8, 1:13)
p(2)	t(1:64, 2:58:8, 1:13)
...	...
p(4)	t(1:64), 8:64:8, 53:64)

001 where the size of the third dimension is 64 and is not divisible by the size of the second
002 dimension of `p`. Then, the sizes of a number of blocks in the third dimension are different.
003

004 3.2.4 Align directive

005 Synopsis

006 The `align` directive specifies that arrays are to be mapped in the manner given by a template.
007

008 Syntax

```
009 [F] !$xmp align array-name ( align-source [, align-source] ... )  
010 with template-name (align-subscript [, align-subscript] ... )  
011  
012 [C] #pragma xmp align array-name [align-source] [[align-source]] ...  
013 with template-name (align-subscript [, align-subscript] ... )  
014
```

015 where *align-source* must be one of:

```
016 scalar-int-variable  
017 *  
018 :  
019
```

020 and *align-subscript* must be one of:

```
021 scalar-int-variable [ ( + | - ) int-expr ]  
022 *  
023 :  
024
```

025 Note that the variable *scalar-int-variable* appearing in *align-source* is referred to as a *eealign*
026 dummy variable”.

027 Description

028 The array specified by *array-name* is aligned with the template specified by *template-name*.
029 Each element indexed by the sequence of *align-source* is aligned with the element of the template
030 indexed by the sequence of *align-subscript*, where *align-source* and *align-subscript* are interpreted
031 as follows:

- 032 1. The first form of *align-source* and *align-subscript* describes an align dummy variable and
033 its (restricted) expression, respectively. The range of the align dummy variable covers all
034 valid index values in the corresponding dimension of the array.
- 035 2. The second form *ee*** of *align-source* and *align-subscript* describes a dummy variable (not
036 an align dummy variable) that does not appear anywhere in the directive.
 - 037 • The second form of the align-source is said to *eecollapse*” the corresponding dimension
038 of the array. As a result, the index along the corresponding dimension makes no
039 difference in determining the alignment.
 - 040 • The second form of the align-subscript is said to *ee replicate*” the array. Each ele-
041 ment of the array is replicated and aligned to all index values in the corresponding
042 dimension of the template.
- 043 3. Both *align-source* and *align-subscript* are *ee:*”, then each element of an array is aligned
044 with each element of the template.

Restrictions

- In the sequence of *align-subscript*, the same align variable must not appear more than once.
- In *align-subscript*, an align dummy variable must not appear more than once.
- In *int-expr* of *align-subscript*, an align dummy variable must not appear.
- If either *align-source* or *align-subscript* is `ee*`, then the corresponding *align-source* or *align-subscript* must be `ee*`.

Examples

Example 1

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

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

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

Example 2

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

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

Example 3

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

The array element `a(j)` is replicated and aligned with each template element `t(1,j) ~ t(10,j)`. Assume that the upper and lower bounds of the first dimension of the template `t` are 1 and 10, respectively.

Example 4

```
FORTRAN
!$xmp template t(n1,n2)
      real a(m1,m2)
!$xmp align a(*,j) with t(*,j)
```

The subarray `a(:,j)` is aligned with each template element, `t(1,j) ~ t(n1,j)`.

If `ee*` in the first dimension of array `a` is replaced by dummy variable `i`, and `ee*` in the first dimension of the template is replaced by dummy variable `k`, then we have:

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

3.2.5 Shadow directive

Synopsis

The `shadow` directive declares and allocates a 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 shadow width for which the area is used to communicate the neighbor element of the block of an array specified by *array-name*, which is distributed onto each node. When *shadow-width* is of the form *int-expr* : *int-expr*, the shadow area is added at the upper and lower bounds with the width specified in the specified dimension. When *shadow-width* is a form of *int-expr*, the shadow area is added at the upper and lower bounds with the same width specified in the specified dimension. When *shadow-width* is of the form *ee**, 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. Note that this type of shadow is sometimes referred to as a eefull shadow.”

The data stored in the storage area declared by the `shadow` directive is referred to as a shadow object. The shadow object can be explicitly defined and referenced only by the method described below. Of the data allocated to a storage area other than a shadow area, data representing the same array element as that of a shadow object is referred to as a reflection source of the shadow object. Conceptually, a shadow object and its reflection source are not mapped to the same processor at the same time.

Restrictions

- The value specified by *shadow-width* must be a non-negative integer.

3.2.6 `template_fix` directive

Synopsis

This directive is an executable directive that fixes the shape of the 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`

001 where *dist-format* is one of:

```
002 *  
003 block  
004 cyclic [( int-expr )]  
005 gblock ( int-array )  
006
```

007 Description

008 The `template_fix` directive is an executable directive to fix the shape of the template, which is
009 initially undefined, by specifying the sizes of each dimension and the distribution format at run-
010 time. The array aligned to an initially undefined template must be an allocatable array, which
011 cannot be allocated until the template is fixed by the `template_fix` directive. Any executable
012 directives that have such a template in their `on` clause must not be executed before the template
013 is fixed by the `template_fix` directive. Any undefined template can be fixed only once by the
014 `template_fix` directive.

015 Note that the meaning of *dist-format* in the `distribute` clause is the same as that in the
016 `distribute` directive.

017 Restrictions

- 018 • When the `template_fix` directive is executed, the template specified by *template-name*
019 must be undefined.
- 020 • The sequence of *dist-format* in the `template_fix` directive and the sequence of *dist-format*
021 in the `distribute` directive specified by *template-name* must be identical, except for the
022 brackets following `gblock`.
- 023 • The sequence of either the *template-spec* or `distribute` clause must be given.
- 024 • The `template_fix` directive must appear in executable context.

025 Example

```
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
```

FORTRAN

```
!$xmp template :: t(:)  
!$xmp distribute (gblock(*)) :: t  
  
    real , allocatable :: a(:)  
5 !$xmp align (i) with t(i) :: a  
    ...  
    N = ...; M(...) = ...  
    ...  
!$xmp template_fix(gblock(M)) t(N)  
10  
    ...  
    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 to `t`. After the size `N` of `t` and the mapping array `M` is defined, `t` is fixed by the `template_fix` directive, and `a` is allocated.

3.3 Work mapping construct

3.3.1 Task construct

Synopsis

The `task` construct defines an explicit task that is executed in a specified node.

Syntax

```
[F]  !$xmp task on node-ref — template-ref
      block
      !$xmp end task

[C]  #pragma xmp task on node-ref — template-ref
      block
```

Description

When the execution encounters a `task` construct, a block is executed if the executing node is one of the nodes specified by *nodes-ref* or *template-ref*. Otherwise, the execution of the block is skipped.

This line was inserted by Sakagami for svn test.

If the task is not surrounded by the `tasks` construct, *nodes-ref* and *template-ref* are evaluated at the entry of the block. Otherwise, *nodes-ref* and *template-ref* of the `task` construct are evaluated at the entry of the surrounding `tasks` construct. The results of the evaluation must be the same in every node in the executing node set.

In order to execute the block by the nodes specified by *nodes-ref* and *template-ref*, new executing node sets are created conceptually. The node set executing outside the `task` construct is referred to as the eeparent executing node set”.

Restrictions

- The node set specified by *nodes-ref* or *template-ref* must be a subset of the parent executing node set.

3.3.2 Tasks construct

Synopsis

The `tasks` construct executes multiple tasks in arbitrary order.

Syntax

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

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

Description

The `tasks` construct executes the surrounding `task` constructs (child task) in arbitrary order without implicit synchronization at the entry of each child task. As a result, if there is no overlap between executing node sets of adjacent tasks, these tasks can be executed in parallel.

Conceptually, *nodes-ref* or *template-ref* in the `task` constructs of child tasks are evaluated at the beginning of the `tasks` construct.

No implicit synchronization is performed at the entry of the `tasks` construct.

When a `nowait` clause is specified, implicit synchronization is not performed at the end of the `tasks` construct. Without a `nowait` clause, implicit synchronization, which guarantees the completion of all inter-task communications among the child tasks, is performed in order to ensure that all communications issued inside children tasks are finished.

Example

```

016          _____ FORTRAN _____
017          subroutine caller
018          !$xmp nodes p(1000)
019          real a(100,100)
020
021          ...
022  5  !$xmp tasks
023  !$xmp task on p(1:500)
024      call task1(a)
025  !$xmp end task
026  !$xmp task on p(501:800)
027      call task1(a)
028  10 !$xmp end task
029  !$xmp task on p(801:1000)
030      call task1(a)
031  !$xmp end task
032  15 !$xmp end tasks
033
034      ...
035      end do
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054
055
056
057

```

```

          _____ FORTRAN _____
          subroutine task1(a)
          ...
          !$xmp nodes q(*)
          real a(100,100)
          ...
          end subroutine

```

3.3.3 Loop construct

Synopsis

The `loop` construct specifies that the iterations of a loop will be executed in parallel by threads in nodes of the executing node set. The iterations are distributed across nodes where the specified data is accessed locally. If the loop includes reduction operations, then reduction references must be specified to obtain the correct results.

Syntax

```
[F]  !$xmp loop [ ( loop-index [, loop-index] ... ) ] on on-ref [ reduction-ref ]
[C]  #pragma xmp loop [ ( loop-index [, loop-index] ... ) ] on on-ref [ reduction-ref ]
```

where *on-ref* is one of:

template-ref
nodes-ref.

```

001      reduction-ref is:
002          ( reduction-kind : reduction-spec [ , ... ] )
003
004      reduction-kind is one of:
005          +
006          *
007          .AND.
008          .OR.
009          .EQV.
010          .NEQV.
011          MAX
012          MIN
013          IAND
014          IOR
015          IEOR
016          FIRSTMAX
017          FIRSTMIN
018          LASTMAX
019          LASTMIN
020
021
022      reduction-spec is:
023          reduction-variable [ / location-variable [ , ... ] ) / ]
024
025

```

Description

The `loop` construct is associated with a loop nest consisting of one or more loops that follow the directive and distribute the execution of iterations across nodes in the executing node set. Since the iteration range of the loop for each node is determined before the loop is executed, efficient loop execution can be expected.

When *on-ref* is *template-ref*, according to the distribution of the specified template, the set of loop indices of iterations to be executed in each node is decided, and the iterations are executed by this node set as an executing node set. Therefore, before the `loop` construct is executed, the referenced template must be fixed. When *template-spec* is *ee**, the corresponding dimension is collapsed so that it is ignored for the distribution of the loop. When *template-spec* is *ee:*, the nodes for all of the template elements in the corresponding dimension are assigned to iterations for execution.

When *on-ref* is *nodes-ref*, the node set associated with the loop index is created as an executing node set to execute the iteration of the loop index.

When the loop includes reduction operations, proper *reduction-ref* must be specified in order to obtain semantically correct results, and the reduction operation is executed on the specified local reduction variable just after the execution of the loop.

The loop construct that has *template-ref* as *on-ref* and the `reduction` clause, except in cases with *reduction-kind* of `FIRSTMAX`, `FIRSTMIN`, `LASTMAX`, or `LASTMIN`, is equivalent to the `reduction` construct with the following *template-spec* replacements:

ee:h to *g *h*,

gthe loop index *h* to *gan* iteration range of the loop.

```

051          _____ FORTRAN _____
052          | !$xmp loop (j) on t(:,j) reduction(...)
053          |     do j = js, je
054          |         ...
055          |         do i = 1, N
056          |             ...
057          |

```

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

```
        end do
        ...
    end do
```

This loop with the reduction clause is equivalent to the code shown below:

```

                                FORTRAN
!$xmp loop (j) on t(:,j)
    do j = js, je
        ...
        do i = 1, N
            ...
        end do
        ...
    end do
!$xmp reduction(...) on t(*,js:je)
```

Note that, unlike the `loop` construct with the `reduction` clause, the `reduction` construct does not consider initialization for the reduction variable. The following programs return different values of the `sum` variable after the reduction operation. When `sum` is initialized to zero, these programs return the same results.

```

                                FORTRAN
    sum = 123.45
!$xmp loop (i) on t(i) reduction(+:sum)
    do i = 1, N
        sum = sum + a(i)
    end do

    sum = 123.45
!$xmp loop (i) on t(i)
    do i = 1, N
        sum = sum + a(i)
    end do
!$xmp reduction(+:sum) on t(1:N)
```

Restrictions

- *loop-index* must be the loop index of the loop after the directive or the loop index of the loops nested by the loop.
- When the sequence of the *loop-index* is omitted, the loop index of the loop after the directive is specified.
- *template-spec* appearing in *template-ref* must be either `ee*`, `ee:`, or *loop-index*. In the case of *loop-index*, the loop index must be the loop index of the outer loop of the loop.
- *nodes-ref* must reference different node sets for each *loop-index*. These node sets consist of different nodes. That is, a node must not be included in more than one node set.
- When more than one `loop` construct is nested, *on-ref* of each loop must also be nested.
- The initial value, upper bound, and increment of *loop-index* must be invariant in the loop and the same in all executing nodes.

- If *reduction-kind* is FIRSTMAX, FIRSTMIN, LASTMAX, or LASTMIN, then *reduction-spec* has more than one *location-variable*.
- *reduction-ref* must reference the reduction operations associated with the loop after the directive or the loops nested by the loop.
- *location-variable* must be fixed in the loop after the directive or the loops nested by the loop.
- *reduction-variable* must not be referred at a certain iteration in the loop, except for updating itself.
- *reduction-variable* and *location-variable* must not exist in *reduction-ref* of nested loops.

Examples

Example 1

```

018                                     FORTRAN
019 !$xmp distribute t(block) onto p
020 !$xmp align (i) with t(i) :: a, b
021     ...
022 !$xmp loop (j) on t(i)
023     do i = 1, N
024         a(i) = 1.0
025         b(i) = a(i)
026     end do

```

The loop construct decides the node that executes the iterations, according to the distribution of template *t*, and distributes the execution. This example is equivalent to the example shown below, but will be faster because the iterations executed in each node are decided before executing the loop.

```

034                                     FORTRAN
035 !$xmp distribute t(block) onto p
036 !$xmp align (i) with t(i) :: a, b
037     ...
038     do i = 1, N
039         !$xmp task on t(i)
040             a(i) = 1.0
041             b(i) = a(i)
042         !$xmp end task
043     end do

```

Example 2

```

048                                     FORTRAN
049 !$xmp distribute t(*,block) onto p
050 !$xmp align (*,j) with t(*,j) :: a, b
051     ...
052 !$xmp loop (j) on t(*,j)
053     do j = 1, M
054         do i = 1, N
055             a(i,j) = 1.0
056             b(i,j) = a(i,j)

```

```

001         end do
002     end do
003
004
005
006

```

Since the first dimension of template `t` is collapsed, this loop is distributed only for the second dimension. This example is equivalent to the `task` construct shown below.

```

007                                     FORTRAN
008 !$xmp distribute t(*,block) onto p
009 !$xmp align (*,j) with t(*,j) :: a, b
010     ...
011     do j = 1, M
012 5 !$xmp task on t(*,j)
013         do i = 1, N
014             a(i,j) = 1.0
015             b(i,j) = a(i,j)
016         end do
017 10 !$xmp end task
018     end do
019
020
021

```

Example 3

```

022                                     FORTRAN
023 !$xmp distribute t(block,block) onto p
024 !$xmp align (i,j) with t(i,j) :: a, b
025     ...
026
027 5 !$xmp loop (i,j) on t(i,j)
028     do j = 1, M
029         ...
030         do i = 1, N
031             a(i,j) = 1.0
032             b(i,j) = a(i,j)
033         end do
034 10
035     end do
036     ...
037 end do
038

```

The distribution for the multi-dimensional array in the nested loop can be described using the sequence of *loop-index* in one loop construct. This example is equivalent to the loop shown below, but will run faster because the iterations to be executed are decided outside of the nested loop. Note that the inner template index set is included in the outer template index set.

```

045                                     FORTRAN
046 !$xmp distribute t(block,block) onto p
047 !$xmp align (i,j) with t(i,j) :: a, b
048     ...
049 5 !$xmp loop (j) on t(:,j)
050     do j = 1, M
051         ...
052 10 !$xmp loop (i) on t(i,j)
053         do i = 1, N
054             a(i,j) = 1.0
055             b(i,j) = a(i,j)
056         end do
057

```

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

```
...
end do
```

Example 4

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

The executing node sets are created by referencing 10 nodes by p(:,i) with different values of i in order to execute each of the associated iterations. This example is equivalent to the loop containing task constructs or static tasks/task constructs.

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

```
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
!$xmp task on p(:,3)
call subtask ( 3 )
!$xmp end task
!$xmp task on p(:,3)
call subtask ( 3 )
!$xmp end task
!$xmp task on p(:,3)
call subtask ( 3 )
!$xmp end task
!$xmp end tasks
```

Example 5

```
FORTRAN
...
lb(1) = 1
iub(1) = 10
lb(2) = 11
iub(2) = 25
lb(3) = 26
iub(3) = 50
```

```

001 !$xmp loop (i) on p(lb(i):iub(i))
002     do i = 1, 3
003         call subtask ( i )
004     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 or `static tasks/task` constructs.

```

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

```

```

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

```

```

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

```

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)` in the variable `s` in 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.

```

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

```

FORTRAN
...
amax = -1.0e30
ip = -1
jp = -1
!$xmp loop (i,j) on t(i,j) reduction(firstmax:amax/ip,jp/)

```

```

001         do j = 1, M
002             do i = 1, N
003                 if( 1(i,j) .gt. amx ) then
004                     amx = a(i,j)
005                     ip = i
006                     jp = j
007                 end if
008             end do
009         end do
010     end do
011

```

This loop computes the maximum value of `a(i,j)` in the variable `amax` in each node. Note that the incomplete maximum value in the corresponding node is computed on `amax` without the reduction clause. When this program is executed sequentially without XscalableMP directives, the first indices of the maximum element are obtained in `ip` and `jp`, so that `FIRSTMAX` is specified in the reduction clause. This example cannot be rewritten with the `reduction` construct.

3.3.4 Array construct

Synopsis

The `array` construct divides the task of array assignment among the owners of the array.

Syntax

```

[F]  !$xmp array on template-ref
[C]  (not defined)

```

Description

In FORTRAN, the array assignment can be used instead of the loop of the assignment for each element. This directive executes the array assignment in each node.

Restrictions

- *template-spec* must have shape conformance with the array assignment after the directive.
- If the range in *template-spec* is omitted, all of the ranges are assumed to be specified.
- The `array` construct is collective, and must not be guarded under conditional execution depend on the node index.

Examples

Example 1

```

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.

```

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

```

```

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

```

Example 2

```

                                FORTRAN
!$xmp template t(100,20)
!$xmp distribute t(block,block) onto p
    dimension a(100,20)
!$xmp align (i,j) with t(i,j) :: a
5   ...
!$xmp array on t
    a = 2.0

```

This example is equivalent to the code shown below.

```

                                FORTRAN
!$xmp template t(100,20)
!$xmp distribute t(block,block) onto p
    dimension a(100,20)
!$xmp align (i,j) with t(i,j) :: a
5   ...
!$xmp loop (i,j) on t(i,j)
    do j = 1, 20
        do i = 1, 100
            a(i,j) = 2.0
10        end do
    end do

```

3.4 Global-view communication and synchronization construct

3.4.1 Reflect construct

Synopsis

The `reflect` directive assigns the value of a reflection source to the corresponding shadow object for variables having the shadow attribute.

Syntax

- [F] `!$xmp reflect array-name [, array-name ...]`
- [C] `#pragma xmp reflect array-name [, array-name ...]`

Description

The `reflect` directive copies the value of the reflection source of the reflect-object specified by *array-name* to all shadow objects. This directive may execute the communications.

3.4.2 Gmove construct

Synopsis

The `gmove` construct copies the data of a distributed array in global view.

Syntax

```
[F] !$xmp gmove [in|out] dest = source
[C] #pragma xmp gmove [in|out] dest = source
```

Description

This directive executes a copy operation of the global data array object distributed into nodes. This directive is followed by the assignment statement of the scalar value and array sections. The assignment operation of the array sections of a distributed array may require communication between nodes.

The assignment statement must be a simple assignment without any arithmetic operations.

In XcalableMP, the C language is extended to support array section notation in order to support an assignment of array objects.

The assignment statement must have one of the following patterns:

- Scalar assignment. For example:

```
s1 = s2          ! s1, s2 is a scalar variable
a(3) = b(i, j)   ! a, b are arrays.
```

- Array assignment. The left-hand value must be an array name, array section, or scalar object. For example:

```
a = b           ! a, b are arrays
a(1:10) = b(n:n+9, k) ! left and right are array section
a(1:10) = s2     ! The left is an array section, left is a scalar variable
a(1:10) = b(i, j) ! The left is an array section, left is a scalar object
```

The `gmove` construct must be executed by nodes in the executing node set. The value of scalar objects, the index value, and the range value of the array section in the assignment statement must be the same in every node executing this directive.

When no option is specified, the copy operation is performed collectively by all nodes in the executing node set. In this case, all elements in both the source array and the target array must be distributed on to the executing node set. If the object on the right-hand side is a local object, then the value of the local object must be the same. In this case, the assignment is performed locally, where the object on the left-hand side is distributed. If the object on the left-hand side is a local object and the object on the right-hand side is global, then this operation performs broadcast operation.

If an `in` option is specified, then the node that owns the element of the object on the left-hand side obtains the data on the right-hand side by the remote copy (get) operation. Therefore, the object on the left-hand side must be distributed onto the executing node set.

If an `out` operation is specified, then the node that owns the element of the object on the right-hand side places the data on the left-hand side by the remote copy (put) operation. Therefore, the object on the right-hand side must be distributed onto the executing node set.

If no option is specified, then the copy can be performed by two-side communication. In this case, the receiver side waits for the sender side, resulting in implicit synchronization.

If an `in` or `out` clause is specified, then the copy operation should be performed by one-side communication for remote memory access. Thus, no synchronization is implied. If synchronization between reader and writer is required, then the programmer must perform synchronization explicitly by a `barrier` construct. If the reader and the writer do not belong to the same executing node set, then point-to-point synchronization by `post-wait` directive can be used.

Restrictions

- The `gmove` construct must be executed by all nodes in the executing node set.

Examples

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

FORTRAN	C
<pre>!\$xmp gmove a(:,1:N) = b(:,3,0:N-1)</pre>	<pre>#pragma xmp gmove a[1:N][:] = b[0:N-1][3][:];</pre>

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

FORTRAN	C
<pre>!\$xmp gmove a(:,1:N) = c(k)</pre>	<pre>#pragma xmp gmove a[1:N][:] = c[k]</pre>

Example 3

FORTRAN
<pre>!\$xmp nodes p(4) real a(4), b(4) !\$xmp distribute (block) onto p :: a,b ... 5 !\$xmp task on p(1:2) ! Only p(1), p(2) execute this section ... !\$xmp gmove a(1:2) = b(2:3) ! Communication to outside ... ! of executing node occurs 10 !\$xmp end task</pre>

3.4.3 Barrier construct

Synopsis

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

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

Syntax

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

Description

The barrier operation is performed between the node set specified by an on clause. If an on clause is not specified, then the current executing node set is used. The barrier construct also has the function of ensuring that all of the remote copy operations that are invoked by gmove in/out constructs executed by the node set specified by the on clause are finished.

Restriction

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

3.4.4 Reduction construct

Synopsis

The reduction construct performs a reduction operation between nodes.

Syntax

```
[F] !$xmp reduction( reduction-kind : variable [, variable ] ... ) [on node-ref | template-ref]  
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]  
where reduction-kind is one of:  
+  
*  
-  
&  
|  
^  
&&  
||  
max  
min
```

Description

The **reduction** construct performs the reduction operation specified by *reduction-kind* for the specified local variable in each node of the node set specified by the **on** clause and sets the reduction results to variables in each node. The variable for the reduction operation is referred to as the “reduction variable”. Thus, after the reduction operation, the value of the reduction variable becomes the same.

When *template-ref* is specified in the **on** clause, the operation is performed in a node set that consists of nodes associated with the specified template. Therefore, before the **reduction** construct is executed, the referenced template must be fixed. When *template-spec* is “*”, nodes in the corresponding dimension are ignored for the reduction operation. When *template-spec* is “:”, nodes for all template elements in the corresponding dimension perform the reduction operation.

When *node-ref* is specified in the **on** clause, the operation is performed in the specified node set. Therefore, before the **reduction** construct is executed, the referenced node set must be fixed.

When the **on** clause is omitted, the operation is performed in the current executing node set.

Restrictions

- The variables specified by the sequence of *variable* must either not be aligned or be replicated among nodes of the node set specified by the **on** clause.
- The **reduction** construct is collective 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 the executing nodes.
- The node set specified by the **on** clause must be a subset of the current executing node set.

Examples

Example 1

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

In the first example, the scalar variable **s** is assumed to contain the partial sum of the variable, and the reduction operation calculates the total sum of the variable. The total sum is stored in the variable in each node.

The second example computes the maximum value of the variable **aa** in the node set that consists of nodes associated with the range of the second dimension of template **t**.

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

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

Example 2

```
001
002                                     FORTRAN
003     dimension a(n,n), p(n), w(n)
004     !$xmp align a(i,j) with t(i,j)
005     !$xmp align p(i) with t(i,*)
006     !$xmp align a(j) with t(*,j)
007
008     ...
009     !$xmp loop (j) on t(:,j)
010         do j = 1, n
011             sum = 0
012     !$xmp loop (i) on t(i,j) reduction(+:sum)
013         do i = 1, n
014             sum = sum + a(i,j) * p(i)
015         end do
016         w(j) = sum
017     end do
018
019
```

This code computes the matrix vector product. The **reduction** clause is specified for the loop construct of the inner loop. This example is equivalent to the following program.

```
020                                     FORTRAN
021
022
023     dimension a(n,n), p(n), w(n)
024     !$xmp align a(i,j) with t(i,j)
025     !$xmp align p(i) with t(i,*)
026     !$xmp align a(j) with t(*,j)
027
028     ...
029     !$xmp loop (j) on t(:,j)
030         do j = 1, n
031             sum = 0
032     !$xmp loop (i) on t(i,j)
033         do i = 1, n
034             sum = sum + a(i,j) * p(i)
035         end do
036     !$xmp reduction(+:sum) on t(1:n,j)
037         w(j) = sum
038     end do
039
040
041
```

In this case, 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 loop index of the outer loop (**tt j**) is different from that for the reduction operation (**i**). However, this code can be modified with the **reduction** construct as follows:

```
042                                     FORTRAN
043
044
045     dimension a(n,n), p(n), w(n)
046     !$xmp align a(i,j) with t(i,j)
047     !$xmp align p(i) with t(i,*)
048     !$xmp align a(j) with t(*,j)
049
050     ...
051     !$xmp loop (j) on t(:,j)
052         do j = 1, n
053             sum = 0
054
055
056
057
```

```

001      !$xmp loop (i) on t(i,j)
002 10         do i = 1, n
003             sum = sum + a(i,j) * p(i)
004             end do
005             w(j) = sum
006         end do
007
008 15 !$xmp reduction(+:w) on t(1:n,*)

```

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

3.4.5 Bcast construct

Synopsis

The `bcast` construct executes broadcast communication from one node.

Syntax

```

[F]  !$xmp bcast variable [, variable...] [from nodes-ref [on nodes-ref]template-ref]
[C]  #pragma xmp bcast variable [, variable...] [from nodes-ref [on nodes-ref]template-ref]

```

Description

The value of the variables specified by *variable* is broadcasted from the node specified by the `from` clause (called the source node) to the nodes in the node set specified by the `on` clause. The specified variable for broadcast is referred to as the "broadcast variable". After executing this directive, the value of the broadcast variable becomes the same as the value of the source node. If the `from` clause is omitted, then the first node of the node set is a source node. If the `on` clause is omitted, then the operation is performed in the current executing node set.

Restrictions

- The variables specified by the *variable* sequence must either not be aligned or be replicated among nodes of the node set specified by the `on` clause.
- The `bcast` construct is collective, which means that a `bcast` construct must be executed by all of the executing nodes, and each variable appearing in the construct must have the same value among all of the executing nodes.
- The node set specified by the `on` clause must be a subset of the current executing node set.
- The source node specified by the `from` clause must belong to the node set specified by the `on` clause.

001
002
003
004
005
006
007 **Chapter 4**
008
009

010
011 **Support for local-view programming**
012
013
014

015
016 XcalableMP adopts coarray notations as an extension of languages for local-view programming.
017 In the case of FORTRAN as the base language, most coarray notations are compatible with
018 that of Co-array Fortran (CAF), except that the `task` constructs are used for task parallelism.
019

020 This line was inserted by Sakagami for svn test.
021

022 **4.1 Coarray notation of XcalableMP**
023

024 The coarray is declared and referenced as in CAF. In addition, the notations are extended in
025 XcalableMP so that the coarray declarations can be followed by the `on` clause (*on nodes-ref*). In
026 this case, the images of the coarray are allocated on the node set specified by *nodes-ref*. If the
027 `on` clause is not specified, then the images of the coarray are allocated on the executing node
028 set as a default.
029
030

031 **Examples**
032

033 **Example 1**

```
034 FORTRAN  
035 !$xmp nodes w(50)  
036     real wa(100)[*]  
037     ...  
038 !$xmp tasks  
039 5 !$xmp task on w(1:30)  
040     call task1 ( wa )  
041 !$xmp end task  
042 !$xmp task on w(32:50)  
043     call task2 ( wa )  
044 10 !$xmp end task  
045     ...  
046     subroutine task1 ( aa )  
047 !$xmp nodes w(50)  
048 !$xmp nodes p(50) = *  
049 15     real aa(100)[*] on w  
050     real b(100)[*]  
051     ...  
052     subroutine task2 ( aa )  
053 !$xmp nodes w(50)  
054 20 !$xmp nodes p(20) = *
```

```

001     real aa(100)[*] on w
002     real c(100)[*]
003     ...
004

```

Coarray `wa` mapped onto the entire node set `w` is passed as an argument to subroutines `task1` and `task2`. In these subroutines, the argument is referenced by the parameter `aa`. In this case, the declaration of the coarray for the parameter `aa` requires an `on` clause to specify the executing node set, which is different from the entire node set. By this declaration, the subroutine can access `wa`. Note that coarrays `b` and `c` are local variables in subroutines `task1` and `task2`, respectively, and these coarrays are mapped onto the executing node set indicated by `p`. Since `b` is mapped to `w(1:30)` and `c` is mapped onto `w(31:51)`, there is no way to access coarray `c` (b) from `task1` (`task2`) because a subroutine can access the entire node set or its subset as well as the executing node set.

Example 2

```

018     _____ FORTRAN _____
019     !$xmp nodes w1(20)
020     real one(100)[*]
021     real two(50)[20,*]
022

```

The entire node set has 200 nodes, onto which coarray `one` and `two` are mapped. In this case, it is guaranteed that `one(...)[i+(j-1)*20]` and `two(...)[i,j]` are mapped into the same node. This mapping rule is compatible with Coarray Fortran. This is also the case for the following code. That is, the shape of the entire node set has nothing to coarray.

```

028     _____ FORTRAN _____
029     !$xmp nodes w2(20,10)
030     real one(100)[*]
031     real two(50)[20,*]
032     ...
033
5  OR
034     !$xmp nodes w3(10,5,4)
035     real one(100)[*]
036     real two(50)[20,*]
037     ...
038     ...
039

```

Example 2

```

042     _____ FORTRAN _____
043     !$xmp nodes w1(200)
044     !$xmp nodes w2(20,10)
045     real one(100)[*]
046     real two(50)[20,*]
047     real ichi(100)[*] on w2
5  real ni(50)[20,*] on w2
049

```

In many application programs, neighbor communication is an important communication pattern. The optimal mapping from the logical node to the physical node depends on the number of dimensions. For example, the optimal mapping for the communication between `one(...)[i]` and `one(...)[i+1]` is different from the mapping for the communication between `two(...)[i,j]` and `two(...)[i+1,j]`.

In XcalableMP , several mappings can be described. In this example, the mapping may be compatible with Coarray Fortran, and a different mapping can be selected. By switching the runtime option, node sets `w1` and `w2` can be different node sets. Coarrays `one` and `two` are mapped to node set `w1` (that is, the entire node set), and `one(...)[i+(j-1)*20]` and `two(...)[i,j]` are mapped to the same node. Coarrays `ichi` and `ni` are mapped onto a different node set `w2`, and `ichi(...)[i+(j-1)*20]` and `ni(...)[i,j]` are mapped onto the same physical node. However, it is not guaranteed that a pair of `one(...)[i]` and `ichi(...)[i]` and a pair of `two(...)[i,j]` and `ni(...)[i,j]` are mapped into the same node. Note that this point is not compatible with Coarray Fortran.

4.2 Coarray in the C language

XcalableMP adopts coarray notation. In order to use coarray notation in C , we propose some extensions of the language.

4.2.1 Array section notation

Array section notation is a notation to describe the part of array that is adapted in Fortran90. In C , an array section has the following form:

array-name '[' [*lower-bound*] ':' [*upper-bound*] ':' *step*]' ...

An array section is built from subsets of the elements of an array object – those associated with a selected subset of the index range attached to the object. The *upper-bound* and the *lower-bound* specify the range of array elements of an array object. Either the *upper-bound* or the *lower-bound* can be omitted in the index range of a section, in which case they default to the lowest or highest values taken by the index of the array. Therefore, `A[:]` is a section containing the whole of `A`. If the *step* is specified, then the elements of an array section are every *step*-th element in the specified range. For example, `B[1:10:3]` is an array section of size 4 containing every third element of `B` with indices between 1 and 10 (i.e., indices 1, 4, 7, 10). Collectively, ranges specified by *upper-bound*, *lower-bound*, and *step* are referred to as triplets. For multi-dimensional arrays, some dimensions can be subscripted with a normal scalar expression, and some dimensions can be “sectioned” with triplets.

Example

Array `A` is declared by

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

Then:

<code>A[10:19]</code>	array section of 10 elements from <code>A[10]</code> to <code>A[19]</code>
<code>A[10:]</code>	array section of 90 elements from <code>A[10]</code> to <code>A[99]</code>
<code>A[:9]</code>	array section of 10 elements from <code>A[0]</code> to <code>A[9]</code>
<code>A[10:19:2]</code>	array section of 5 elements from <code>A[10]</code> to <code>A[18]</code> by step 2
<code>A[:]</code>	entire array of <code>A</code>

4.2.2 Assignment of array sections

Array-valued expressions can be used by array section in assignments.

array-section1 = *array-section2*

The expression on the right-hand side of the assignment must be conformable with the array variable on the left-hand side. Thus, both sides must have the same shape, i.e., the same number of dimensions and size of each dimension.

Examples

```
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
```

```
_____ C _____
int A[10];
int B[5];
...
A[5:9] = B[0:4]; // copy the elements from A[5] to A[9],
                // to the elements from B[0] to B[4]
```

4.2.3 Declarations and reference of coarray

A coarray is declared by the coarray directive in C .

```
[C] #pragma xmp coarray coarray-dimension :: array-variable
```

```
[C] #pragma xmp coarray array-variable coarray-dimension
```

The coarray directive declares a coarray with the coarray dimension.

Examples

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

double p[100][100];
#pragma xmp coarray p[20][*]
```

The coarray object is referenced in the following expression:

scalar-variable : *image-index*

array-section-expression : *image-index*

This expression indicates the co-array on the image indicated by *image-index*.

```
_____ C _____
A[:] = B[:]:[10]; // copy from B on image 10 to A
```

4.3 Directive for local-view programming

4.3.1 local_alias directive

Synopsis

The `local_alias` directive declares the local data object as an alias to the data of a distributed array allocated in each node.

Syntax

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

```
[C] #pragma xmp local_alias :: local-array-name => global-array-name
```

Description

This directive declares that the 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 part of the corresponding global array that is owned by each node. Note that the local part includes the shadow area.

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.

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*.
- [F] The array specified by *local-array-name* must be a deferred-shape array.
- [C] The array specified by *local-array-name* must be a pointer.

Examples

Example 1

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

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

      real :: b
10 !$xmp local_alias :: b => a
```

Array `a` is distributed by block onto four nodes. Node `n(2)` has a local array of twenty-five elements (`a(25:50)`) with shadow areas of size one on the upper and lower bounds. 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`.

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

Example 2

```
001                                     FORTRAN
002
003 !$xmp nodes n(4)
004 !$xmp template :: t (100)
005 !$xmp distribute (cyclic) onto n :: t
006
007
008 5      real :: a (100)
009 !$xmp align (i) with t(i) :: a
010 !$xmp shadow (1) :: a
011
012      real :: b(0:)
013 10 !$xmp local_alias :: b => a
014
```

An array `a` is distributed cyclically onto four nodes. Node `n(2)` has a local array of twenty-five elements (`a(2:100:4)`) with shadow areas of size one on the upper and lower bounds. The lower bound of local alias `b` is declared to be zero. As a result, `b` is an array of 27 elements (`b(0:26)`) on `n(2)`. The table below shows the correspondence of each element of `a` to that of `b`.

a	b
lower shadow	0
2	1
6	2
10	3
...	...
98	25
upper shadow	26

Example 3

```
034                                     FORTRAN
035
036 !$xmp nodes n(4)
037 !$xmp template :: t(:)
038 !$xmp distribute (block) onto n :: t
039
040 5      real , allocatable :: a(:)
041 !$xmp align (i) with t(i) :: a
042
043      real :: b(:)[*]
044 !$xmp local_alias :: b => a
045
046 10     c
047
048
049 !$xmp template_fix :: t(128)
050
051 15     allocate (a(128))
052
053     if (me < 4) b(4) = b(4)[me +1]
054
```

Since the global array `a` is an allocatable array, its local alias `b` is not defined when the subroutine starts execution. `b` is defined when `a` is allocated at the `allocate` statement.

Note that `b` is declared as a coarray and therefore can be accessed in the same manner as a normal coarray.

4.3.2 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 )
```

Restriction

- `nodes-ref` must represent one node.
- `tag` must be an integer expression.

Description

This construct prohibits statements that follow the 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 `post` construct issued with the arguments of `tag` and `nodes-ref` 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) on the node specified by `nodes-ref`.

Example

Example 1

FORTRAN	FORTRAN
S1 !\$xmp post (p(2), 1)	!\$xmp wait (p(1), 1) S2

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

The `post` construct on the left match the `wait` construct on the right because each `nodes-ref` represents each other and both `tags` 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

FORTRAN
!\$xmp wait S3

It is assumed that this code is executed by node `p(3)`.

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

4.3.3 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 ] ) ]
```

Restriction

- *nodes-ref* must represent one node.
- *tag* must be an integer expression.

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 *tag* and *nodes-ref* on a node (referred to as a *waiting node*) dynamically matches at most one `post` construct issued with the arguments of the waiting node and the same value as *tag* on the node specified by *nodes-ref*.

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

4.3.4 Critical construct

Synopsis

The `critical` construct restricts execution of the associated structured block to a single node at any given time.

Syntax

```
[F] !$xmp critical  
    block  
    !$xmp end critical  
  
[C] #pragma xmp critical  
    block
```

Description

This construct specifies a *critical region*, at the beginning of which a node waits until no other node is executing it.

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

Procedure call and data mapping for procedure 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 6

Runtime library

`xmp_get_node_num()`

`xmp_get_num_nodes()`

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

Other issues

7.1 Hybrid programming with MPI and OpenMP

7.1.1 MPI

7.1.2 OpenMP

7.2 Interface to numerical libraries

7.2.1 ScaLAPACK

Chapter 8

Sample Programs

Example 1

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

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

double u[XSIZE+2][YSIZE+2],
      uu[XSIZE+2][YSIZE+2];
10 #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()
15 {
    int x,y,k;
    double sum;
    for(k = 0; k < NITER; k++){
        /* old <- new */
20 #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
25 #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;
30    }
    m */
    sum = 0.0;
    #pragma xmp loop on t[x] reduction(+:sum)
    for(x = 1; x <= XSIZE; x++)
35    for(y = 1; y <= YSIZE; y++)
```

```

001         sum += (uu[x][y]-u[x][y]);
002 #pragma xmp task on master
003         printf("sum = %g\n",sum);
004     }
005
006

```

Example 2

```

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

```

```

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

double pvt_v[N]; // local
10
/* gaussian elimination with partial pivoting */
dgefa(double a[n][LDA],int lda, int n,int ipvt,int *info)
#pragma xmp align a[:,i] to t(i)
{
15     REAL t;
    int idamax(),j,k,kp1,l,nm1,i;
    REAL x_pvt;

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

25
        /* if (a[k][l] != ZERO) */
#ifdef XMP
#pragma xmp gmove
        pvt_v[k:n-1] = a[l][k:n-1];
30 #else
        for(i = k; i < n; i++) pvt_v[i] = a[i][l];
#endif

        /* interchange if necessary */
35         if (l != k){
#ifdef XMP
#pragma xmp gmove
        a[l][:] = a[k][:];
40 #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];

```

```

001 #endif
002 45     }
003     /* compute multipliers */
004     t = -ONE/pvt_v[k];
005     A_dscal(k+1, n-(k+1),t,a[k]);
006
007
008     /* row elimination with column indexing */
009     for (j = kp1; j < n; j++) {
010         t = pvt_v[j];
011         A_daxpy(k+1,n-(k+1),t,a[k],a[j]);
012     }
013 }
014 55     }
015     ipvt[n-1] = n-1;
016 }
017
018 dgesl(double a[n][LDA],int lda,int n,int pvt[n],double b,int job)
019 60 #pragma xmp align a[:] [i] to t(i)
020 #pragma xmp align b[i] to t(i)
021 {
022     REAL t;
023     int k,kb,l,nm1;
024
025 65     nm1 = n - 1;
026     /* job = 0 , solve a * x = b, first solve l*y = b */
027     for (k = 0; k < nm1; k++) {
028         l = ipvt[k];
029 70 #pragma xmp gmove
030         t = b[l];
031         if (l != k){
032 #pragma xmp gmove
033         b[l] = b[k];
034 75 #pragma xmp gmove
035         b[k] = t;
036     }
037     A_daxpy(k+1,n-(k+1),t,a[k],b);
038 }
039 }
040
041 80     /* now solve u*x = y */
042     for (kb = 0; kb < n; kb++) {
043         k = n - (kb + 1);
044 #pragma xmp task on t(k)
045 85 {
046         b[k] = b[k]/a[k][k];
047         t = -b[k];
048     }
049 #pragma xmp bcast t from t(k)
050     A_daxpy(0,k,t,a[k],b);
051 }
052 }
053 }
054 }
055 }
056 }
057 }

```

```

001      /*
002      95 * distributed array based routine
003      */
004      A_daxpy(int b,int n,double da,double dx[n],double dy[n])
005      #pragma xmp align dx[i], dy[i] to t(i)
006      {
007
008      100     int i,ix,iy,m,mp1;
009             if(n <= 0) return;
010             if(da == ZERO) return;
011             /* code for both increments equal to 1 */
012             #pragma xmp loop on t(b+i)
013             105     for (i = 0;i < n; i++) {
014                     dy[b+i] = dy[b+i] + da*dx[b+i];
015             }
016     }
017
018
019     110 int A_idamax(int b,int n,double dx[n])
020     #pragma xmp align dx[i] to t(i)
021     {
022         double dmax, g_dmax;
023         int i, ix, itemp;
024         115     if(n == 1) return(0);
025
026
027         /* code for increment equal to 1 */
028         itemp = 0;
029         dmax = 0.0;
030     120 #pragma xmp loop on t(i) reduction(lastmax:dmax/itemp/)
031         for (i = b; i < n; i++) {
032             if(fabs((double)dx[i]) > dmax) {
033                 itemp = i;
034                 dmax = fabs((double)dx[i]);
035             }
036     125     }
037     }
038     return (itemp);
039 }
040
041
042     130 A_dscal(int b,int n,double da,double dx[n])
043     #pragma xmp align dx[i], dy[i] to t(i)
044     {
045         int i;
046         if(n <= 0)return;
047
048     135
049         /* code for increment equal to 1 */
050         #pragma xmp loop on t(i)
051         for (i = b; i < n; i++)
052             dx[i] = da*dx[i];
053     140 }
054
055
056
057

```

Index

Executable, 10

Nodes, 11

align, 18

array, 30

barrier, 33

bcast, 37

critical, 45

declarative directive, 10

distribute, 15

gmove, 32

local_alias, 41

loop, 23–27, 36

post, 44, 45

reduction, 24, 25, 34–36

reflect, 31

shadow, 20

tasks, 22

task, 22, 23, 29, 35

template_fix, 20

template, 14

wait, 44, 45

align, 18, 19, 21, 26

align dummy variable, 18

alignment, 9

array, 15, 30

barrier, 14, 15, 34

bcast, 14, 37

broadcast variable, 37

coarray, 38, 41

critical, 45

data mapping, 8

Directive

- Executable, 10
- Nodes, 11
- align, 18
- array, 30
- barrier, 33
- bcast, 37
- critical, 45
- declarative directive, 10
- distribute, 15
- gmove, 32
- local_alias, 41
- loop, 23–27, 36
- post, 44, 45
- reduction, 24, 25, 34–36
- reflect, 31
- shadow, 20
- tasks, 22
- task, 22, 23, 29, 35
- template_fix, 20
- template, 14
- wait, 44, 45

directive, 10

distribute, 13, 16, 17, 21, 26

distribution, 9

end task, 14, 23, 38

end tasks, 14, 23, 38

entire node set, 7

entire set of nodes, 7

Example

- align, 19, 21, 26
- array, 15, 30
- barrier, 14, 15
- bcast, 14
- coarray, 41
- distribute, 13, 17, 21, 26
- end tasks, 14, 23, 38
- end task, 14, 23, 38
- gmove, 33
- loop, 13, 15, 26
- nodes, 12–14, 17, 38
- post-wait, 44
- reduction, 14, 15, 35
- tasks, 14, 23, 38
- task, 13–15, 23, 38
- template_fix, 21
- template, 17, 21

executing node array, 8

executing node set, 7

001 executing nodes, 7
 002
 003 global data, 8
 004 gmove, 32, 33
 005
 006 image index, 8
 007
 008 Laplace, 49
 009 Linpack, 50
 010 local, 8
 011 local data, 9
 012 local_alias, 41
 013 loop, 13, 15, 23, 26
 014
 015 node, 6, 11
 016 node array, 7
 017 node number, 7
 018 node reference, 13
 019 node set, 7
 020 nodes, 12–14, 17, 38
 021 non-local, 8
 022
 023
 024 parent executing node set, 22
 025 post, 44
 026 post-wait, 44
 027
 028 reduction, 14, 15, 34, 35
 029 reduction variable, 35
 030 reflect, 31
 031 replicated execution, 8
 032
 033
 034 Sample Program
 035 Laplace, 49
 036 Linpack, 50
 037 shadow, 9, 20
 038 shadow object, 20
 039 Syntax
 040 align, 18
 041 array, 30
 042 barrier, 34
 043 bcast, 37
 044 coarray, 41
 045 critical, 45
 046 directive, 10
 047 distribute, 16
 048 gmove, 32
 049 local_alias, 41
 050 loop, 23
 051 node reference, 13
 052 node, 11
 053 post, 44
 054 reduction, 34
 055 reflect, 31
 056 shadow, 20
 057 tasks, 22
 task, 22
 template reference, 15
 template_fix, 20
 template, 14
 wait, 45
 task, 8, 13–15, 22, 23, 38
 tasks, 14, 22, 23, 38
 template, 8, 14, 17, 21
 template reference, 15
 template_fix, 20, 21
 wait, 45
 work mapping, 8