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 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 . . . 4
2.4 Interactions between global-view and local-view . . . 5
2.5 Execution model and Task . . . 5
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 C language . . . 40
4.2.1 Array sections . . . 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 local_alias directive 41
4.3.2 Post directive 43
4.3.3 Wait directive 43
4.3.4 Critical directive 43

5 Procedure Call and Data mapping for procedure argument 44

6 Runtime Library 45

7 Other issues 46
7.1 Hybrid Programming with MPI and OpenMP 46
7.1.1 MPI 46
7.1.2 OpenMP 46
7.2 Interface to numerical libraries 46
7.2.1 ScaLAPACK 46

8 Sample Programs 47

List of Figures

2.1 Hardware Model 3
2.2 Parallelization by 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, runtime library routines that can be used to specify distributed-memory parallel programming in C and FORTRAN program. These define the specification of XcalableMP Application Program Interface (XcalableMP API). This specification provides a model of parallel programming for distributed memory multiprocessor systems. The directives extend the C and FORTRAN base languages as to describe distributed memory parallel program.

1.1 Scope

XcalableMP API is used to explicitly specify the action to be taken by the compiler and runtime system to execute the parallel program in 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

Features of XcalableMP are summarized as follows:

Language extensions for familiar languages C and FORTRAN , which 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 parallelizing the original sequential code using minimal modification with simple **directives**, like OpenMP .

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

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

For flexibility and extensibility, the execution model allows **combining 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 the experiences of HPF , Fujitsu XPF (VPP FORTRAN) and OpenMPD.

Chapter 2

Overview of XcalableMP model and language

2.1 Hardware model and execution model

The target of XcalableMP is a distributed memory system (Figure 2.1). Each compute 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 the other nodes via communication. It is however assumed that accessing remote memory is much slower than the access of local memory.

The basic execution model of XcalableMP is a SPMD (Single Program Multiple Data) 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, the synchronization and communication occurs between nodes. That is, no synchronization and communications happen without directives. In this case, the program does duplicated 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 one thread executes XcalableMP directives at a time.

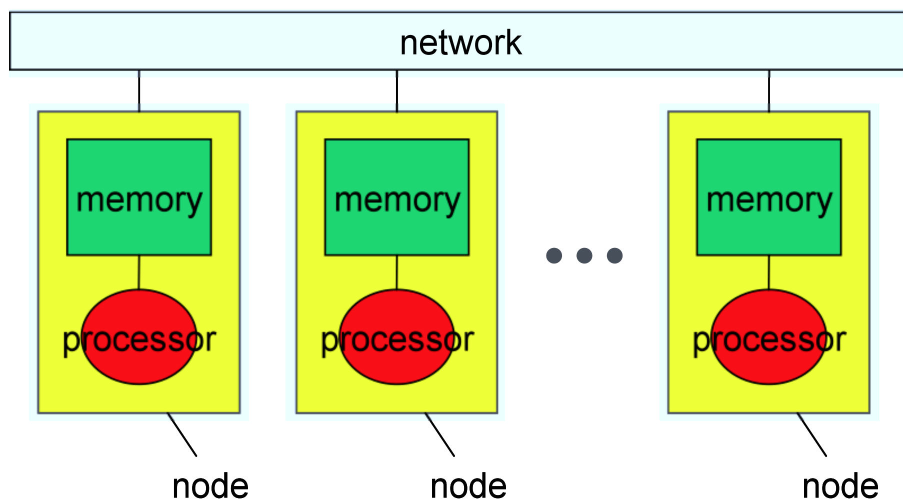


Figure 2.1: Hardware Model

As default, data declared in the program is allocated in each node, and is referenced locally by threads executed in the node.

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

In contrast to the local-view programming mode, a global-view programming model is one in which programmers express their algorithm and data structure as a whole, mapping them to the node set. The programmers describe the data distribution and the work mapping to express how to distribute data and share the work among nodes. The variables in global-view programming model look like a shared memory spanning over nodes.

2.2 Global-view programming model

The global-view programming model is useful when, starting from sequential version of the program; the programmer parallelizes it in data-parallel model by adding directives incrementally with minimum modifications.In the global-view programming model, the programmer describes the data distribution of data shared among the nodes by data distribution directives. `loop` construct maps works in iterations to the node where computed data is located. Global-view communication directives are used to synchronize between nodes, keep the consistency of shadow area, and move a part of distributed data globally. It should be noted that the programmer must keep all data reference required computations done locally by any appropriate directives.

In many cases, the XcalableMP program using the global-view is based on a sequential program and able to produce the same result independent from the number of compute nodes (Figure 2.2). The global view provides a programming model in which computation and data are distributed onto compute nodes.

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

Data Mapping

Specify Data distribution and mapping to nodes (partially inherited from HPF)

Work Mapping (Parallelization)

Assign works to nodes. `loop` construct map each iteration to nodes where the referenced elements are located, and Task construct execute each task parallel in different node set.

Communication and Synchronization

Describe how to communicate and synchronize with the other compute nodes. In XcalableMP , the inter-node communication must be described explicitly. The compiler guarantees that communication takes place only if communication is explicitly specified.

2.3 Local-view programming model

Local view is suitable for the programs explicitly describing the algorithm of each node and explicit remote data reference (Figure 2.3). As MPI is considered to have the local view, the local view programming model of XcalableMP has high interoperability with MPI.

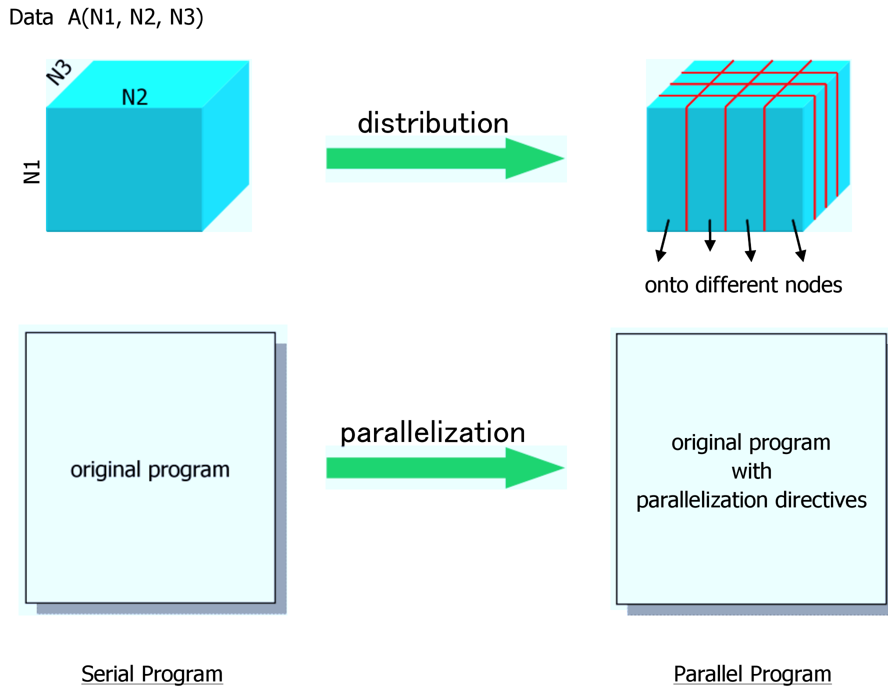


Figure 2.2: Parallelization by global-view programming model

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, 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 the communication to get the value takes place. If the access is updating the value, then the communication to put a new value takes place.

2.4 Interactions between global-view and local-view

The node in XcalableMP is used to distribute data or distribute computational load. In 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 global-view and local-view of data.

Data may have a global view and local view and can be accessed from a global view or local view. XcalableMP provides some directives to put local name (alias) to the global data declared in global-view programming mode so that the data can be referenced also in local-view programming model. It may be useful to optimize the program by explicit remote data reference in 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 called entire nodes.

A task is a specific instance of executable code and its data environments executed in a set of nodes. A task when starting a program in entire nodes is called an initial task. The initial task can generate a subtask which executes on a subset of the nodes by `task` construct. A set

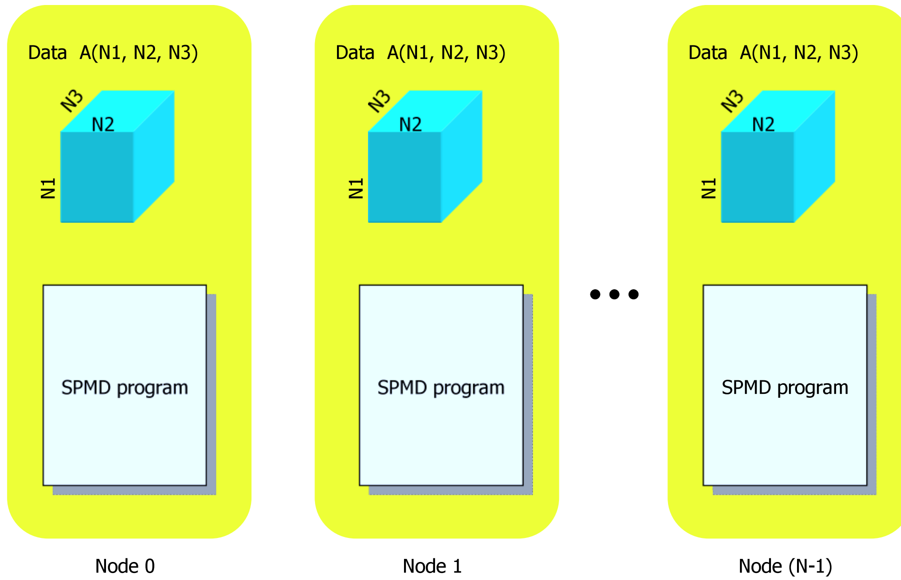


Figure 2.3: Local-view programming model

of nodes executing the same task is called executing nodes. If no `task` construct is encountered, a program is executed as one task, and its executing nodes are entire nodes.

As far as no directives are encountered, a program executes locally. When the same codes are executed, almost same computation is performed in each node, called duplicated execution. When the threads encounter `loop` construct or `array` construct, the specified loop is executed in parallel so that each iteration is assigned to the node where specified data element is located.

A new task is generated by `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 communications between nodes, a set of directives are provided. In local view programming model, a coarray features are adopted for remote data reference. It should be noted that all synchronization and communication are specified by directives explicitly, and without such directives, no communications are executed implicitly by the compiler.

2.6 Glossary

node

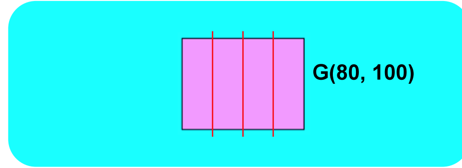
A compute node, which may have several cores sharing main memory, has its own local memory. in distributed memory system. Each node is connected via network. An XcalableMP program begins as a single thread of execution in each node.

node number

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

```
      dimension L(50, 40)                         ! local view (default)
      dimension G(80, 100)                        ! global view
!$xmp template, distribute (block) onto (0:4) ::  T(100)
!$xmp align with T :: G
```

Global name space (virtual)



Data allocation

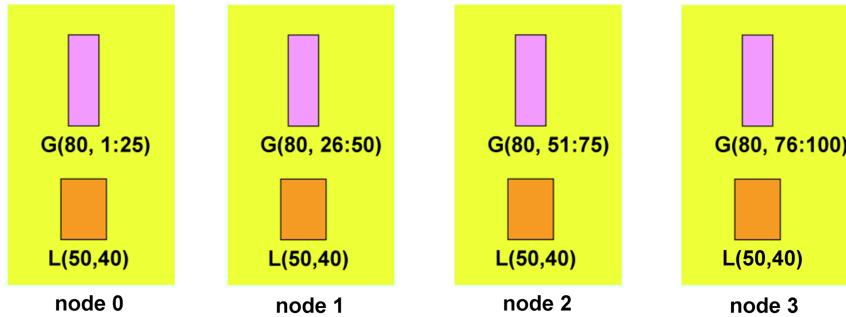


Figure 2.4: Global view and local view

node set

A set of nodes

entire node set, entire node

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

executing node set, executing nodes

A node set executing a certain region of program. The executing node set which execute a whole program is a entire node set. The executing node set of a task is a node set which executes the task.

node array

A multi-dimensional array containing nodes. Node array have a name and shape as it attributes.

executing node array

Node array which contain executing node.

task

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

template

A dummy array used to express an index space associated with an array. It is also used to describe a iteration space of loop. Template has a name, dimension and lower and upper bound of each dimension as attributes.

replicated execution

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

data mapping

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

work mapping

Assignment of iterations to nodes in parallel loop, and tasks to nodes.

image index

A number assigned to each images of coarray. The value is equal to or larger than 1. Note that the image index of the coarray mapping to entire nodes is identical to the node number.

local

Execution of a program has side-effect only on data in the node. In this case, no communication to other nodes takes place.

non-local

Execution of a program requires communication to other nodes, and has some side-effect to other nodes.

global data

Data declared as a distributed array and shared by nodes.

local data

Data is allocated in each node, and referenced only within the node.* collective An operation must be executed by every node in the executing node set to perform an operation working 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 mapping element 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 data and template.

shadow

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

Chapter 3

Directives

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

`xxx`	`type-face` characters are used for literal.
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 by using `#pragma` mechanism provided by C standards. In FORTRAN , XcalableMP directives are specified by using special comments that are identified by unique sentinels `!$xmp`. The rules of FORTRAN directives in fixed source format and free source format follow these in OpenMP and HPF.

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

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

Directives are classified into declarative directives and executable directives. The `declarative directive` is one that my 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 declaration statements in the base language. For example, node declarations by `node` directives are effective from declared point to the end of the file in C, and within subprogram in FORTRAN .

`Executable` directives are placed in executable context. A stand-alone directive is an executable directive which has no associated user code, such as a `barrier` directive. Some executable directives compose directive construct with associated user code, as in following format:

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

Note that in `loop` construct, `end` can be omitted.

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

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

In C, statement can be a compound-statement.

3.1 Node Declaration

3.1.1 `Nodes` Directive

Synopsis

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

Syntax

[F] `!$xmpnodes` [(*map-type*)] *nodes-name*(*nodes-size*) [,*nodes-name*(*nodes-size*)]...
[F] `!$xmpnodes` [(*map-type*)] *nodes-name*(*nodes-size*) [,*nodes-name*(*nodes-size*)]...=*
[F] `!$xmpnodes` [(*map-type*)] *nodes-name*(*nodes-size*) [,*nodes-name*(*nodes-size*)]...=*nodes-ref*

[C] `#pragma xmpnodes` [(*map-type*)] *nodes-name*(*nodes-size*) [,*nodes-name*(*nodes-size*)]...
[C] `#pragma xmpnodes` [(*map-type*)] *nodes-name*(*nodes-size*) [,*nodes-name*(*nodes-size*)]...=*
[C] `#pragma xmpnodes` [(*map-type*)] *nodes-name*(*nodes-size*) [,*nodes-name*(*nodes-size*)]...=*nodes-ref*

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

Description

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

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

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

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

Restrictions

- *nodes-name* is an identifier in class(1), and must not conflict to other names in class(1).
- In FORTRAN , the second form cannot be used in main program and module.
- *nodes-size* can be "*" only in the last dimension.
- Node name referenced in *nodes-ref* must not be a reference to *nodes-name* directly or indirectly.
- If nodes-size does not contain "*" the product of all *nodes-size* must be same as the total size of referenced node set. The referenced node set is the entire node in the first form, executing node set in the second form, and reference node set by *nodes-ref* in the third form.

- nodes-subscript in *nodes-ref* must not be "*".

Examples

These are examples of the first and third form 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. As `regular` is not specified, it is not guaranteed that `Ar(1)` and `p(3)` is the same node and the node number of `z(1,1)` is 1.

FORTRAN

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

C

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

Example using regular option. Since node array `p` is declared without 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 regular option have the order in which `q(1,1), q(2,1), q(3,1), q(4,1), q(1,2), ...` have the node number 1,2,3,4,5, ... And in the node array z with regular option, `z(1,1), z(2,1), z(1,2), z(2,2), z(1,3), z(2,3), ...` have the node number 1, 2, 3, 4, 5, 6, ...

FORTRAN

```
      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
```

Example using in subprogram. Since the node declaration has the second form, the caller of the subroutine `foo` must be executed with 16 nodes. The declaration for the node array `q` with 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

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

3.1.2 Node reference

Synopsis

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

Syntax

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

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

Description

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

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

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

The symbol "*" in *nodes-subscript* in a subarray of a node array specifies a subscript associated to the executing node in the node array of the executing node set. Thus, the following node reference by name with k-th subscript "*"

$$\mathtt{p}(s_1,\ \ldots,\ s_{k-1},\ *,\ s_{k+1},\ \ldots,\ s_n) \text{ where subscripts } s_i \text{ except } s_k \text{ must not "*"}$$

is evaluated at the node

$$\mathtt{p}(j_1,\ \ldots,\ j_{k-1},\ j_k,\ j_{k+1},\ \ldots,\ j_n) \text{ where } j_i \text{ is an integer}$$

into

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

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

Examples

Assumed that `p` is a nodes name and `m` is an integer variable.

- Target node array by `distribute` directive

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

- Target subarray of node array in `nodes` directives

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

- In `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 `loop` directive, sets of executing nodes are respectively specified for the iterations.
  `!$xmp loop (i) on p(lb(i):lb(i+1)-1)`
- In `barrier` directive and `reduction` directive, executing nodes are specified.
  ```
  !$xmp barrier on p(5:8)
  !$xmp reduction (+:a) on p(*,:)
  ```
- In `bcast` directive, a source node and executing nodes are specified.
  `!$xmp bcast b from p(k) on p(:)`

Examples

FORTRAN
```
      subroutine caller
!$xmp nodes p(1000)
      real a(100,100)
      ...
!$xmp tasks
!$xmp  task on p(1:500)
        call task1(a)
!$xmp  end task
!$xmp  task on p(501:800)
        call task1(a)
!$xmp  end task
!$xmp  task on p(801:1000)
        call task1(a)
!$xmp  end task
!$xmp end tasks
      ...
      end do
```

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

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 ":", then the shape of the template is initially undefined. This template must not be referenced until the shape is defined by *template_fix* directives at run-time. If *int-expr* is specified as *template-spec*, the default lower bound is 1.

Restrictions

- Each *template-spec* must be either all [*int-expr* :] *int-expr* or all ":".

3.2.2 Template reference

Synopsis

Template reference expression is used to reference a subset of the referenced template.

Syntax

template-ref	*template-name* [(*template-subscript* [, ...])]
template-subscript	*int-expr* — *triplet* — `*`

Description

Template reference refers a subarray of the template array.

The subscript of subarray of a template array must be either an integer, a triplet or "`*`". The notation of subarray using a triplet in subscript is same as one in FORTRAN .

Examples

Assumed that `t` is a template name.

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

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

- In `loop` directive, an element of the template at which each loop iteration is aligned.

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

- In `array` directive, a template which the following array assignment statement is aligned with.

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

- In `barrier`, `reduction` and `bcast` directives, executing nodes are specified indirectly.

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

3.2.3 Distribute directive

Synopsis

The `distribute` directive specifies a distribution of templates.

Syntax

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

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

Where *dist-format* must be one of:

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

Description

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

The interpretation of dist-format is as follows:

`*` The corresponding dimension is not distributed.

`block` The corresponding dimension of the template is divided into the same size of contiguous block which are distributed onto the corresponding dimension of the node set. Let `d` the size of the corresponding dimension of the template and let `p` the size of the corresponding dimension of node sets. If `d mod p` is not zero, then the dimension of the template is divided into `d/ceil(d/p)` blocks with size `ceil(d/p)` and one block with size `d%ceil(d/p)`, and each block is assigned in sequence to an index along the corresponding dimension of the node set. Note that if `k=p-d/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 the contiguous blocks with size `n`, and blocks are distributed on to the corresponding dimension of node set in a round-robin manner.

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

If more than one `gblock(*)` are specified in *dist-format*, 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 "*" must be equal to the rank of the node set specified by *nodes-name*.
- The array *int-array* in the parentheses following `gblock` must be an integer one-dimensional array, and its size must be equal to the size of the corresponding dimension of node set.
- The element of the array *int-array* in the parentheses following `gblock` must be non-negative integer.

- The sum of the element of the array *int-array* in the parentheses following `gblock` must be equal to or larger 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 with 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 with cyclic format of size 8, 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` with cyclic format. The third dimension is distributed onto the second dimension of node set `p` with block format. It results as shown in the following.

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)

Where the size of the third dimension is 64 and not divisible by the size of the second dimension of `p`, then some blocks in the third dimension have different size.

3.2.4 Align directive

Synopsis

The `align` directive specifies that arrays are to be mapped in the same way as certain template.

Syntax

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

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

Where *align-source* must be one of:

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

`align-subscript` must be one of:

`scalar-int-variable` [(`+` | `-`) *int-expr*]
`*`
`:`

Note that the variable *scalar-int-variable* appearing in *align-source* is called "align dummy variable".

Description

The array specified by *array-name* is aligned with the template specified by *template-name*. Each element indexed by the sequence of *align-source* is aligned to the element of template indexed by the sequence of *align-subscript*. Where *align-source* and *align-subscript* are interpreted as follows:

1. The first form of *align-source* and *align-subscript* describes an align dummy variable and its (restricted) expression respectively. The range of an align dummy variable covers over all valid index values in the corresponding dimension of the array.
2. The second form "*" of *align-source* and *align-subscript* describes a dummy variable (not an align dummy variable) which does not appear anywhere in the directive.
   - The second form of align-source is said to "collapse" the corresponding dimension of the array. As a result, the index along the corresponding dimension makes no difference in determining the alignment.
   - The second form of align-subscript is said to "replicate" the array. Each element of the array is replicated, and is aligned to all index values in the corresponding dimension of the template.
3. Both of *align-source* and *align-subscript* is ":", then each element of an array is aligned to 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 of *align-source* or *align-subscript* is ":", then the corresponding *align-source* or *align-subscript* must be ":".

Examples

Example 1

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

The array element `a(i)` is aligned to 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 to 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 to each template element `t(1,j)` ∼ `t(10,j)`, suppose the upper bound and lower bound 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 to each template element, `t(1,j)` ∼ `t(n1,j)`.

It is interpreted as follows, if "*" in the first dimension of the array `a` is replaced by a dummy variable `i`, and "*" in the first dimension of the template is replaced by a dummy variable `k`.

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

3.2.5 Shadow directive

Synopsis

`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 of which area is used to communicate the neighbor element of block of an array specified by *array-name* which is distributed on to each node. When *shadow-width* is the form of *int-expr* : *int-expr*, the shadow area is added at upper bound and lower bound with specified width in the specified dimension. When *shadow-width* is a form of *int-expr*, the shadow area is added at upper bound and lower bound with the same specified width in the specified dimension. When shadow-width is the form of "`*`", the whole 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 such type of shadow is sometimes called "full shadow."

The data stored in the storage area declared by the `shadow` directive is called a shadow object. The shadow object can be explicitly defined and referenced only by the below-described method. 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 called a reflection source of the shadow object. Conceptually, a shadow object and its reflection source are not mapped to one 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 which 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*

Where *dist-format* is one of:

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

Description

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

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

Restrictions

- When `template_fix` directive is executed, the template specified by template-name must be undefined.
- The sequence of *dist-format* in `template_fix` directive and the sequence of *dist-format* in the `distribute` directive specified by *template-name* must be identical except brackets followed by `gblock`.
- Either the sequence of *template-spec* or `distribute` clause must be given.
- `template_fix` directive must appear in executable context.

Example

FORTRAN

```
!$xmp template :: t(:)
!$xmp distribute (gblock(*)) :: t

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

Since the shape is `(:)`, and 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 `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 which 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.

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

To execute the block by nodes specified by *nodes-ref* and *template-ref*, new executing node sets are created conceptually. The node set executing outside the `task` construct is called "parent executing node set".

Restrictions

- The node set specified by nodes-ref or template-ref must be a subset of 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 surrounded 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 task constructs of child tasks are evaluated at the beginning of the tasks construct.

No implicit synchronization is done at the entry of tasks construct.

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

Example

FORTRAN

```
      subroutine caller
!$xmp nodes p(1000)
      real a(100,100)
      ...
!$xmp tasks
!$xmp  task on p(1:500)
        call task1(a)
!$xmp  end task
!$xmp  task on p(501:800)
        call task1(a)
!$xmp  end task
!$xmp  task on p(801:1000)
        call task1(a)
!$xmp  end task
!$xmp end tasks
      ...
      end do
```

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, reduction references must be specified to obtain 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.

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

reduction-kind is one of:
`+`
`*`
`.AND.`
`.OR.`
`.EQV.`
`.NEQV.`
`MAX`
`MIN`
`IAND`
`IOR`
`IEOR`
`FIRSTMAX`
`FIRSTMIN`
`LASTMAX`
`LASTMIN`

reduction-spec is:
reduction-variable [/ *location-variable* [, ...]) /]

Description

The `loop` construct is associated with a loop nest consisting of one or more loops that follow the directive, and distribute execution of iterations across nodes in the executing node set. As 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 these node set as an executing node set. Therefore, before the `loop` construct is executed, the referenced template must be fixed. When *template-spec* is "*", the corresponding dimension is collapsed so that it is ignored for distribution of loop. When *template-spec* is ":", nodes for all 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 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 `reduction` clause except in cases with *reduction-kind* of `FIRSTMAX`, `FIRSTMIN`, `LASTMAX` or `LASTMIN`, is equivalent to a `reduction` construct with following *template-spec* replacements:
g:h to g *h,
gthe loop indexh to ga iteration range of the looph.

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

```
        ...
    end do
```

This loop with 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 the `reduction` construct does not care of initialization for the reduction variable unlike the `loop` construct with `reduction` clause. The following programs return different values of variable `sum` after the reduction operation. When `sum` is initialized to zero, they return 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 *loop-index* is omitted, it is interpreted that the loop index of the loop after the directive is specified.
- *template-spec* appearing in *template-ref* must be either "*", ":" or *loop-index*. In case of *loop-index*, the loop index must the loop index of 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 more than one node sets.
- When more than one `loop` constructs are 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 same in all executing nodes.

- If *reduction-kind* is `FIRSTMAX`, `FIRSTMIN`, `LASTMAX` or `LASTMIN`, *reduction-spec* has more than one *location-variable*.

- *reduction-ref* must reference reduction operations associated 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 updating itself.

- *reduction-variable* and *location-variable* must not exist in *reduction-ref* of nested loops.

Examples

Example 1

FORTRAN

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

The `loop` construct decide a node which executes the iterations, according to distribution of template t, and distribute the execution. The example is equivalent to the example shown below, while it will be faster because executed iterations in each node is decided before executing the loop.

FORTRAN

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

Example 2

FORTRAN

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

```
        end do
      end do
```

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

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

Example 3

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

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

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

```
      ...
      end do
```

Example 4

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 10 nodes reference by `p(:,i)` with different the value of `i` in order to execute each 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
```

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

The executing node sets with different size are created by `p(lb(i):iub(i))` with different value of i for imbalanced work loads. This example is equivalent to the loop containing `task` constructs, or static `tasks/task` constructs.

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)` into the variable `s` in each node. Note that only the partial sum is computed on `s` without reduction clause. This example is equivalent to the code 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/)
```

```
      do j = 1, M
          do i = 1, N
              if( 1(i,j) .gt. amx ) then
                  amx - a(i,j)
                  ip = i
                  jp = j
              end if
          end do
      end do
```

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

3.3.4 Array Construct

Synopsis

The `array` construct divides the work of array assignment among the owner 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 assignment for each element. This directive executes the array assignment in each node.

Restrictions

- *template-spec* must be 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.

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

Example 2

FORTRAN
```
!$xmp template t(100,20)
!$xmp distribute t(block,block) onto p
      dimension a(100.20)
!$xmp align (i,j) with t(i,j) :: a
      ...
!$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)
!$xmp align (i,j) with t(i,j) :: a
      ...
!$xmp loop (i,j) on t(i,j)
      do j = 1, 20
         do i = 1, 100
            a(i,j) = 2.0
         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 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 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 global data array object distributed into nodes. This directive is followed by the assignment statement of 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 , C language is extended to support array section notation to support an assignment of array objects.

The assignment statement must have one of 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 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. And the value of scalar objects, and index value, range value of array section in the assignment statement must be 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 source array and target array must be distributed on to the executing node set. If the object in right hand side is local object, the value of the local object must be same. In this case, the assignment is done locally where the object in left hand side is distributed. And, if the object in left hand side is a local object and the object in right hand side is global, then this operation performs broadcast operation.

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

If an `out` operation is specified, the node which owns the element of the object in right hand side puts the data in left hand side by remote copy (put) operation. Therefore, the object in right hand side must be distributed on to the executing node set.

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

If an `in` or `out` clause is specified, the copy operation should be done by one-side communication for remote memory access. Thus, no synchronization is implied. If synchronization is

required between reader and writer, the programmer must do synchronization explicitly by a `barrier` construct. If reader and 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 left hand side and right hand side are distributed array, then the copy operation can be performed by all-to-all communication. If the left hand side is a duplicated array, this copy is performed by multi-cast communication. If the right hand side is a duplicated array, no communication is required.

FORTRAN

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

C

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

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

FORTRAN

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

C

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

Example 3

FORTRAN

```
!$xmp nodes p(4)
      real a(4), b(4)
!$xmp distribute (block) onto p :: a,b
      ...
!$xmp task on p(1:2)        ! Only p(1), p(2) execute this section
      ...
!$xmp gmove
      a(1:2) - b(2:3)        ! Communication to outside
      ...                  ! of executing node occurs
!$xmp end task
```

3.4.3 Barrier Construct

Synopsis

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

Syntax

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

001 **Description**
002
003 The barrier operation is performed between the node set specified by an on clause. If on clause
004 is not specified, the current executing node set is used. The barrier construct also has a function
005 of ensuring that all of the remote copy operations that are invoked by gmove in/out constructs
006 executed by the node set specified by the on clause are finished.
007
008 **Restriction**
009
010 - The node set specified by `on` clause must be a subset of the executing node set.
011
012
013 ### 3.4.4 Reduction Construct
014
015 **Synopsis**
016
017 The `reduction` construct performs a reduction operation between nodes.
018
019 **Syntax**
020
021 `[F] !$xmp reduction(` *reduction-kind : variable* [, *variable*] ... `)` [`on` *node-ref*|*template-ref*]
022 Where *reduction-kind* is one of:
023 `+`
024 `*`
025 `.AND.`
026 `.OR.`
027 `.EQV.`
028 `.NEQV.`
029 `MAX`
030 `MIN`
031 `IAND`
032 `IOR`
033 `IEOR`
034
035 `[C] #pragma xmp reduction(` *reduction-kind : variable* [, *variable*] ... `)` [`on` *node-ref*|*template-ref*]
036 Where *reduction-kind* is one of:
037 `+`
038 `*`
039 `-`
040 `&`
041 `|`
042 `^`
043 `&&`
044 `||`
045 `max`
046 `min`
047
048
049 **Description**
050
051 The `reduction` construct performs the reduction operation specified by *reduction-kind* for the
052 specified local variable in each node of the node set specified by `on` clause, and set the reduction
053 results to variables in each node. The variable for the reduction operation is called "reduction
054 variable". Thus, after a reduction operation, the value of the reduction variable becomes same.
055
056 When *template-ref* is specified in `on` clause, the operation is done in a node set that consists
057 of nodes associated with the specified temp late. 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 corresp onding dimension perform the reduction operation.

When *node-ref* is specified in `on` clause, the operation is done in the specified node set. Therefore, before the `reduct ion` construct is executed, the referenced node set must be fixed.

When `on` clause is omitted, the operation is done 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, which it 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 `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(*,:)

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

In the first example, the scalar variable `s` is assumed to contain the partial sum, 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 which consists of nodes associated with all range of the second di mension of template `t`.

In the third example, the minimum value of the variable `bb` in the node set 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

FORTRAN

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

```
          do i = 1, n
              sum = sum + a(i,j) * p(i)
          end do
          w(j) = sum
      end do
```

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

FORTRAN

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

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 operaion (`i`). However this code can be modified with the `reduction` construct as follows:

FORTRAN

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

This code executes a reduction operation on the array `w`, just once which may runs faster.

3.4.5 Bcast Construct

Synopsis

The `bcast` construct execute 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 form the node specified by `from` clause (called source node) to the nodes in the node set specified by `on` clause. The specified variable for broadcast is called "broadcast variable". After executing this directive, the value of the broadcast variable became same as the value of the source node. If `from` clause is omitted, the first node of the node set is a source node. If `on` clause is omitted, the operation is done 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 `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 `on` clause must be a subset of the current executing node set.
- The source node specified by `from` clause must belong to the node set specified by `on` clause.

Chapter 4

Support for Local View Programming

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

4.1 Coarray notation of XcalableMP

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

Examples

Example 1

FORTRAN

```
!$xmp nodes w(50)
      real wa(100)[*]
      ...
!$xmp tasks
!$xmp task on w(1:30)
      call task1 ( wa )
!$xmp end task
!$xmp task on w(32:50)
      call task2 ( wa )
!$xmp end task
      ...
      subroutine task1 ( aa )
!$xmp nodes w(50)
!$xmp nodes p(50) = *
      real aa(100)[*] on w
      real b(100)[*]
      ...
      subroutine task2 ( aa )
!$xmp nodes w(50)
!$xmp nodes p(20) = *
```

```
      real aa(100)[*] on w
      real c(100)[*]
      ...
```

Coarray `wa` mapped on to the entire node set `w` is passed as an argument to subroutine `task1` and `task2`. In the subroutines, the argument is referenced by the parameter `aa`. In this case, the declaration of coarray for the parameter `aa` requires on clause to specify the executing node set which is different from the entire node. By this declaration, the subroutine can access to `wa`. It should be noted that coarray `b` and `c` are local variable in subroutine `task1` and `task2` respectively, and these coarray is mapped on to the executing node set indicated by `p`. Since `b` is mapped to `w(1:30)` and `c` is mapped on to `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, and the executing node set.

Example 2

FORTRAN

```
!$xmp nodes w1(20)
      real one(100)[*]
      real two(50)[20,*]
```

The entire node set has 200 nodes, on to which coarray one and two 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 to Coarray Fortran. This is same in the following code. That is, the shape of the entire node have nothing to coarray.

FORTRAN

```
!$xmp nodes w2(20,10)
      real one(100)[*]
      real two(50)[20,*]
      ...
OR
!$xmp nodes w3(10,5,4)
      real one(100)[*]
      real two(50)[20,*]
      ...
```

Example 2

FORTRAN

```
!$xmp nodes w1(200)
!$xmp nodes w2(20,10)
      real one(100)[*]
      real two(50)[20,*]
      real ichi(100)[*] on w2
      real ni(50)[20,*] on w2
```

In many application programs, the neighbor communication is one of important communication patterns. The optimal mapping from logical node to 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 , one can describe several mappings. In this example, the mapping may be compatible to Coarray Fortran, and different mapping can be selected. By switching runtime option, node set `w1` and `w2` can be different node sets. Coarray `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. Coarray `ichi` and `ni` are mapped on to different node set `w2`, and `ichi(...)[i+(j-1)*20]` and `ni(...)[i,j]` are mapped to the same physical node. But it is not guaranteed that a pair of `one(...)[i]`, `ichi(...)[i]`, a pair of `two(...)[i,j]` and `ni(...)[i,j]` are mapped into the same node. It should be noted that this point is not compatible to Coarray Fortran.

4.2 Coarray in C language

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

4.2.1 Array sections

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

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

An array section is built from some subset of the elements of an array object – those associated with a selected subset of the index range attached to the object. The *lower-bound* and *upper-bound* specify the range of array elements of an array object. Either the *lower-bound* or the *upper-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 array's index. So `A[:]` is a section containing the whole of `A`. If the *step* is specified, 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 (ie, indices 1, 4, 7, 10). Collectively ranges specified by *lower-bound*, *upper-bound* and *step* are referred to as triplets. For multi-dimensional arrays, some dimensions could be subscripted with a normal scalar expression, and some could be "sectioned" with triplets.

Example

Array `A` is declared by

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

Then:

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

4.2.2 Assignment of Array sections

One can use array-valued expressions 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, that is, number of dimension and size of each dimension.

Examples

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 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 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 becomes defined when the corresponding global array becomes 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 aloocated.

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 (cyclic) onto n :: t

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

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

The node `n(2)` has a local array of twenty-five elements (`a(2:100:4)`) with shadow areas of size one on the lower and upper bounds. The lower bound of the local alias `b` is declared to be zero. As a result, `b` is an array of twenty-seven 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 2

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

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

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

      c

!$xmp template_fix :: t(128)

      allocate (a(128))

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

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

4.3.2 Post directive

4.3.3 Wait directive

4.3.4 Critical directive

Chapter 5

Procedure Call and Data mapping for procedure argument

Chapter 6

Runtime Library

`xmp_get_node_num()`

`xmp_get_num_nodes()`

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

FORTRAN

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

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

lap_main()
{
 int x,y,k;
 double sum;
for(k = 0; k < NITER; k++){
        /* old <- new */
#pragma xmp loop on t(x)
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            uu[x][y] = u[x][y];
#pragma xmp reflect uu
#pragma xmp loop on t(x)
        for(x = 1; x <= XSIZE; x++)
          for(y = 1; y <= YSIZE; y++)
            u[x][y] = (uu[x-1][y] + uu[x+1][y] +
                  uu[x][y-1] + uu[x][y+1])/4.0;
    }
m */
    sum = 0.0;
#pragma xmp loop on t[x] reduction(+:sum)
    for(x = 1; x <= XSIZE; x++)
        for(y = 1; y <= YSIZE; y++)
```

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

Example 2

FORTRAN

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

double pvt_v[N];  // local

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Index

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

align, 18, 19, 21, 26
align dummy variable, 18
alignment, 9
array, 15, 30

barrier, 14, 15, 33
bcast, 14, 37
broadcast variable, 37

coarray, 38, 41

data mapping, 8
Directive
    Executable, 10
    Nodes, 11
    align, 18
    array, 30
    barrier, 33
    bcast, 37
    declarative directive, 10
    distribute, 15
    gmove, 32
    local_alias, 41
    loop, 23–27, 36
    reduct ion, 35
    reduction, 24, 25, 34, 36
    reflect, 31
    shadow, 20
    tasks, 22
    task, 22, 23, 29, 35
    template_fix, 20
    template, 14
directive, 10
distribute, 13, 16, 17, 21, 26
distribution, 8

end task, 14, 23, 38
end tasks, 14, 23, 38
entire node, 7
entire node set, 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, 14, 15, 26
    nodes, 12–14, 17, 38
    reduction, 14, 15, 35
    tasks, 14, 23, 38
    task, 13–15, 23, 38
    template_fix, 21
    template, 17, 21
executing node array, 7
executing node set, 7
executing nodes, 7

global data, 8
gmove, 32, 33

image index, 8

Laplace, 47

Linpack, 48
local, 8
local data, 8
local_alias, 41
loop, 14, 15, 23, 26

node, 6, 11
node array, 7
node number, 6
node reference, 13
node set, 7
nodes, 12–14, 17, 38
non-local, 8

parent executing node set, 22

reduction, 14, 15, 34, 35
reduction variable, 34
reflect, 31
replicated execution, 8

Sample Program
  Laplace, 47
  Linpack, 48
shadow, 9, 20
shadow object, 20
Syntax
  `align`, 18
  `array`, 30
  `barrier`, 33
  `bcast`, 37
  `coarray`, 41
  `directive`, 10
  `distribute`, 16
  `gmove`, 32
  `local_alias`, 41
  `loop`, 23
  `node reference`, 13
  `node`, 11
  `reduction`, 34
  `reflect`, 31
  `shadow`, 20
  `tasks`, 22
  `task`, 22
  `template reference`, 15
  `template_fix`, 20
  `template`, 14

task, 7, 13–15, 22, 23, 38
tasks, 14, 22, 23, 38
template, 8, 14, 17, 21
template reference, 15
template_fix, 20, 21

work mapping, 8