XcalableMP programming model and language

Hitoshi Murai, Masahiro Nakao, and Mitsuhisa Sato

Abstract XcalableMP (XMP) is a directive-based language extension of Fortran and C for distributed-memory parallel computers, and can be classified as a partitioned global address space (PGAS) language. One of remarkable characteristics of XMP is that it supports both of global- and local-view parallel programming. This chapter describes the programming model and language specification of XMP.

1 Introduction

Distributed-memory systems are generally used for large-scale simulations. To program such systems, Message Passing Interface (MPI) is widely adopted. However, the programming with MPI is difficult because programmers must describe inter-process communications with consideration of the execution flow of the programs, which might cause deadlocks or wrong results.

To address this issue, a parallel language named High Performance Fortran (HPF) was proposed in 1991. With HPF, users can execute their serial programs in parallel by inserting minimal directives into them. If users specify data distribution with HPF directives, compilers do all other tasks for parallelization (e.g. communication generation and work distribution). However, HPF was not widely accepted eventually because the compiler's automatic processing prevents users from performance tuning

Hitoshi Murai
RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan e-mail: h-murai@riken.jp

Masahiro Nakao
RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan, e-mail: masahiro.nakao@riken.jp

Mitsuhisa Sato
RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan e-mail: msato@riken.jp

and the performance depends heavily on the environment (e.g. compiler version and hardware)

Note: For more detail, please refer: Ken Kennedy, Charles Koelbel and Hans Zima: The Rise and Fall of High Performance Fortran: An Historical Object Lesson, Proc. 3rd ACM SIGPLAN History of Programming Languages Conf. (HOPL-III), pp. 7-1-7-22 (2007).

In such circumstance, to develop a new parallel programming model that enables easy parallelization of existing serial programs and design a new language based on it, “the XMP Specification Working Group” was established in 2008. This group utilized the lessons from the experience of HPF to define a parallel language XcalableMP (XMP). The group was reorganized to one of the working groups of PC Cluster Consortium in 2011.

It is learned from the lessons of HPF that more automatic processing of compilers increases the gap between a program and its execution, and, as a result, decreases the usability of the language.

In XMP, users specify explicitly the details of parallel programs on the basis of compiler directives to make their execution easy-to-understand. In particular, users can specify explicitly communication, synchronization, data distribution, and work distribution to facilitate performance tuning. In addition, XMP supports features for one-sided communication on each process, which was not available in HPF. This feature might enable users to implement parallel algorithms easily.

1.1 Target Hardware

The target of XcalableMP is distributed-memory multicomputers (Figure 1). Each computation node, which may contain several cores, has its own local memory (shared by the cores, if any), and is connected with the others via an interconnection network. Each node can access its local memory directly and remote memory (the memory of another node) indirectly (i.e., through inter-node communication). However, it is assumed that accessing remote memory may be much slower than accessing local memory.

1.2 Execution Model

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

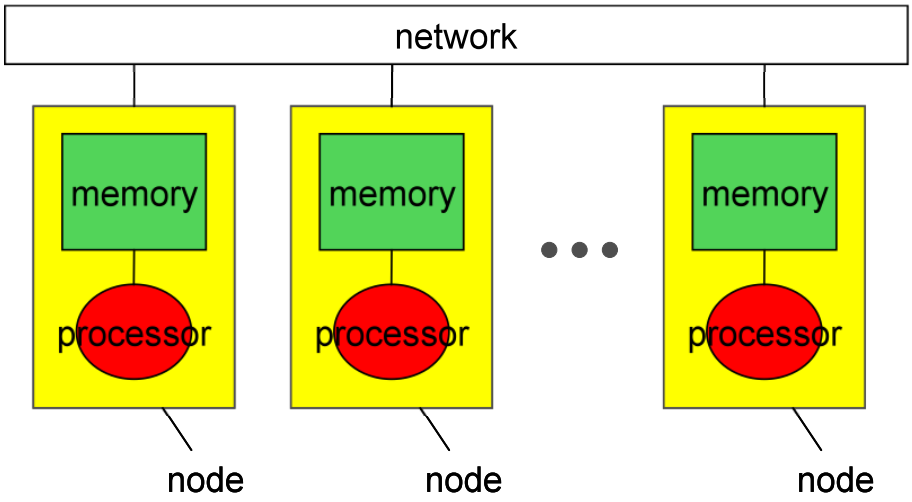


Fig. 1 Hardware model.

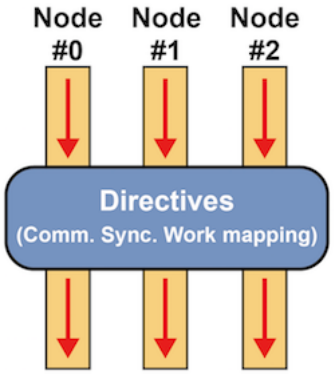
A set of nodes that executes a procedure, statement, loop, a block, etc. is referred to as its *executing node set*, and is determined by the innermost `task`, `loop`, or `array` directive surrounding it dynamically, or at runtime. The *current executing node set* is an executing node set of the current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

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

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

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

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



1.3 Data Model

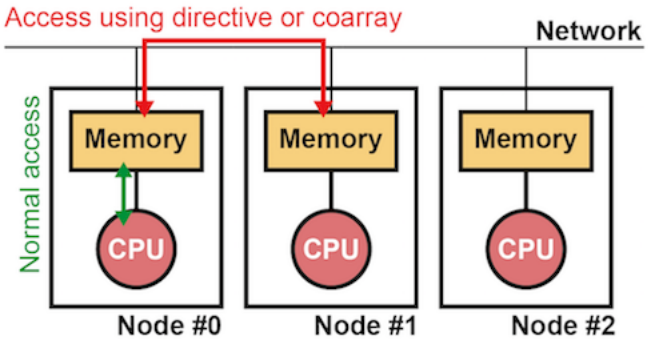
There are two classes of data in XcalableMP: *global data* and *local data*. Data declared in an XcalableMP program are local by default.

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

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

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

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



1.4 Programming Models

1.4.1 Partitioned Global Address Space

Based on the above models that XMP follows, XMP can be classified as a partitioned global address space (PGAS) programming model, such as Co-Array Fortran [1], Unified Parallel C [2], and Chapel [3].

In the PGAS model, multiple executing entities (i.e. threads, processes, or nodes in XMP) share a part of their address space, which is, however, partitioned and a portion of which is local to each executing entity.

The two programming models that XMP supports to achive high performance and productivity on PGAS are explained below.

1.4.2 Global-view Programming Model

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

In many cases, the XcalableMP program according to the global-view programming model is based on a sequential program, and it can produce the same results, regardless of the number of nodes (Figure 2).

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

Data Mapping

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

Work Mapping (Parallelization)

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

Communication and Synchronization

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

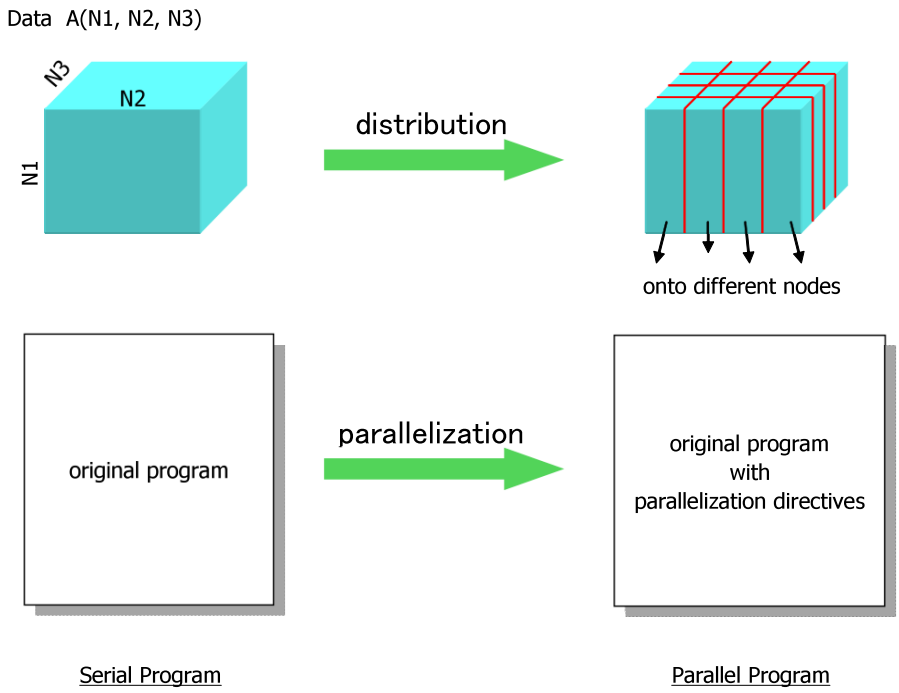


Fig. 2 Parallelization using the global-view programming model.

1.4.3 Local-view Programming Model

The local-view programming model is suitable for programs that explicitly describe an algorithm and a remote data reference that are to be executed by each node (Figure 3).

For the local-view programming model, some language extensions and directives are provided. The coarray notation, which is imported from Fortran 2008, is one such extension, and can be used to specify which replica of a local data is to be accessed. For example, the expression of `A(i)[N]` is used to access an array element of `A(i)` located on the node `N`. If the access is a reference, then a one-sided communication to get the value from the remote memory (i.e., the *get* operation) is issued by the

executing node. If the access is a definition, then a one-sided communication to put a value to the remote memory (i.e., the *put* operation) is issued by the executing node.

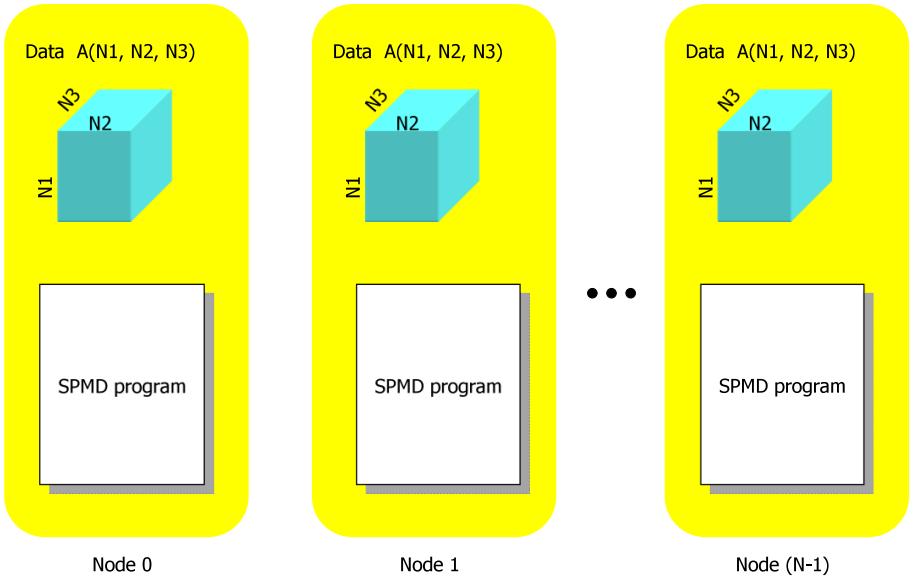


Fig. 3 Local-view programming model.

1.4.4 Interactions between Global View and Local View

In the global view, nodes are used to distribute data and works. In the local view, nodes are used to address data in the coarray notation. In application programs, programmers should choose an appropriate data model according to the structure of the program. Figure 4 illustrates the global view and the local view of data.

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

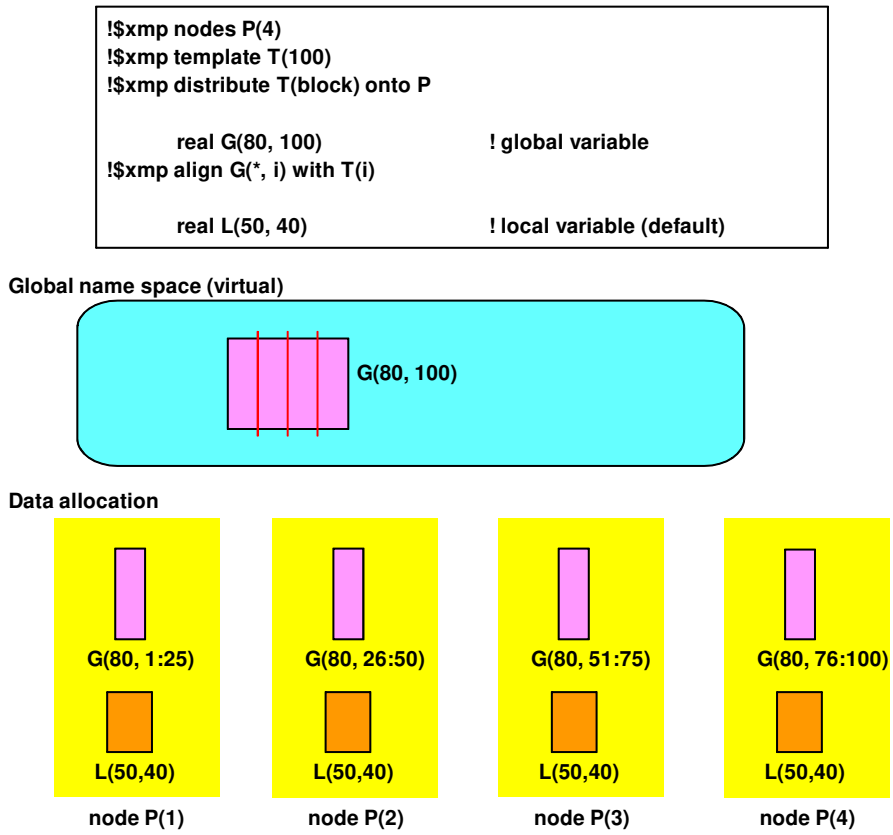


Fig. 4 Global view and local view.

1.5 Base Languages

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

1.5.1 Array Section in XcalableMP C

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

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

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

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

Example

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

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

some array sections can be specified as follows:

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

1.5.2 Array Assignment Statement in XcalableMP C

The value of each element of the result of the right-hand side expression is assigned to the corresponding element of the array section on the left-hand side. When an operator or an elemental function (see section 1.5.4) is applied to array sections in the right-hand side expression, it is evaluated to an array section that has the same shape as that of the operands or arguments, and each element of which is the result of the operator or function applied to the corresponding element of the operands or arguments. A scalar object is assumed to be an array section that has the same shape as that of the array section(s), and where each element has its value.

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

Examples

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

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

1.5.3 Dynamic Allocation of Global Data in XcalableMP C

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

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

Note that the first argument of `xmp_malloc` is the *descriptor* of the global array, which can be obtained with an operator `xmp_desc_of`.

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

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

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

1.5.4 Support of Intrinsic/Built-in Functions of the Base Language

This section describes how intrinsic/built-in functions of the base languges work on XMP's global arrays.

Many other intrinsic and library procedures, such as mapping inquiry functions and transformational procedures, are defined in XMP. See the language specification for their detail.

Array Intrinsic Functions in XcalableMP Fortran

The array intrinsic functions of the base language Fortran are classified into three classes: *inquiry*, *elemental*, and *transformational*.

It is specified as follows how these functions work in the XMP/F programs when a global array appears as an argument.

- Inquiry functions
  The inquiry functions with a global array or its subobject being an argument are regarded as inquiries about the global array, and return its "global" properties as if it were not distributed.
- Elemental functions
  The result of the elemental functions with a global array or its subobject being an argument has the same shape and mapping as the argument. Note that such a reference of these elemental functions is in effect limited to be in the `array` construct.
- Transformational functions
  It is unspecified how the transformational functions work when a global array or its subobject appears as an argument. A processor shall detect such a reference of these functions and issue a warning message for it. Some intrinsic transformational subroutines are defined in section **??** as alternatives to these transformational functions.

Built-in Elemental Functions in XcalableMP C

Some built-in elemental functions that can operate each element of array arguments are defined in XcalableMP C. Such a built-in function accepts one or more array sections as its arguments and returns an array-valued result having the same shape and mapping as the argument. The values of the elements of the result are the same as what would have been obtained if the scalar function of the C standard library had been applied separately to the corresponding elements of each array argument.

These functions may appear on the right-hand side of an array assignment statement, and it should be preceded by the `array` directive if the array section is distributed.

Table 1 shows the list of built-in elemental functions in XcalableMP C. Their elementwise behavior is the same as those of the corresponding functions in the C standard library.

1.6 Interoperability

Most of existing parallel applications are written with MPI. It is not realistic to port them over to XMP because each of them consists of millions of lines.

Because XMP is interoperable with MPI, users can develop an XMP application by modifying a part of an existing one instead of rewriting it totally. Besides, when developing a parallel application from scratch, it is possible to use XMP to write a complicated part of, for example, domain decomposition while they use MPI, which

Table 1 Built-in elemental functions in XcalableMP C. (The first line refers to the element type of their argument(s) and return value.)

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

could be faster than XMP, to write a hot-spot part that need to be tuned carefully. In addition, XMP is interoperable with OpenMP and Python (see Chapter **??**).

It might be difficult to develop an application with just one programming language or framework since it generally has its own strong and weak points. Thus, an XMP program is interoperable with those in other languages to provide both high productivity and performance.

2 Data Mapping

2.1 `nodes` Directive

The `nodes` directive declares the name of a node set and its shape. A node set can have a multi-dimensional shape.

2.1.1 One-dimensional Node Set

XcalableMP C

```
#pragma xmp nodes p[4]
```

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

The nodes directive declares 1-dimensional node set p which has four nodes. In XMP/C, the node set consists of p[0], p[1], p[2], and p[3]. In XMP/Fortran, the node set consists of p(1), p(2), p(3), and p(4).

2.1.2 Multi-dimensional Node Set

XcalableMP C
```
#pragma xmp nodes p[2][3]
```

XcalableMP Fortran
```
!$xmp nodes p(3,2)
```

The nodes directive declares 2-dimensional node set p which has six nodes. In XMP/C, the node set consists of p[0][0], p[0][1], p[0][2], p[1][0], p[1][1], and p[1][2]. In XMP/Fortran, the node set consists of p(1,1), p(2,1), p(3,1), p(1,2), p(2,2), and p(3,2).

Note: The ordering of the elements in a node set depends on the base language, C and Fortran.

2.1.3 Dynamic Node Set

XcalableMP C
```
#pragma xmp nodes p[*]
```

XcalableMP Fortran
```
!$xmp nodes p(*)
```

An asterisk symbol can be used in the nodes directive to declare a dynamic node set. The program declares one-dimensional dynamic node set p by using an asterisk symbol. The size of a dynamic node set is determined at runtime (at the beginning of the execution). For example, when the user runs the sample program with three nodes, the node set p will have three nodes.

The user also can declare multi-dimensional dynamic nodes with an asterisk symbol.

XcalableMP C
```
#pragma xmp nodes p[*][3]
```

XcalableMP Fortran
```
!$xmp nodes p(3,*)
```

When the user runs the sample program with 12 nodes, the node set p will have a shape of [4][3] in C, and (3,4) in Fortran.

Note: The user can use only one asterisk symbol in the last dimension of the node set.

Hint: The dynamic node set may interfere with compiler optimizations. Static node sets may achieve better performance in general.

2.1.4 Partial Node Set

The user can declare a partial node set from the existing node set. Partial node sets can be used to optimize inter-node communication by reducing the number of nodes participating in the communication.

XcalableMP C

```
#pragma xmp nodes p[16]
#pragma xmp nodes q[8]=p[0:8]
#pragma xmp nodes r[4][2]=p[8:8]
```

XcalableMP Fortran

```
!$xmp nodes p(16)
!$xmp nodes q(8)=p(1:8)
!$xmp nodes r(2,4)=p(9:16)
```

In line 1, a node set p which has 16 nodes is declared. In line 2, a partial node set q from the first half of p is declared. In line 3, a 2-dimensional partial node set r from the latter half of p is declared.

The user can declare an 1-dimensional node set from a multi-dimensional node set.

XcalableMP C

```
#pragma xmp nodes p[4][2]
#pragma xmp nodes row[4]=p[:][*]
#pragma xmp nodes col[2]=p[*][:]
```

XcalableMP Fortran

```
!$xmp nodes p(2,4)
!$xmp nodes row(4)=p(*,:)
!$xmp nodes col(2)=p(:,*)
```

In line 1, a two-dimensional node set p which has 4x2 nodes is declared. In line 2, a partial node set row from a single row node set of p is declared. In line 3, a partial node set col from a single column node set of p is declared.

The colon symbols used in the sample program are triplets which indicate that all elements in the dimension are used to declare the target partial node set. The asterisk symbols indicate that the current executing node will be used to declare the target partial node set. For example, col[2] is p[0][0:2] on node p[0][0]/p[0][1] and

is p[1][0:2] on node p[1][0]/p[1][1] in XMP/C. Likewise, col(2) is p(1:2,1) on node p(1,1)/p(2,1) and p(1:2,2) on node p(1,2)/p(2,2) in XMP/Fortran.

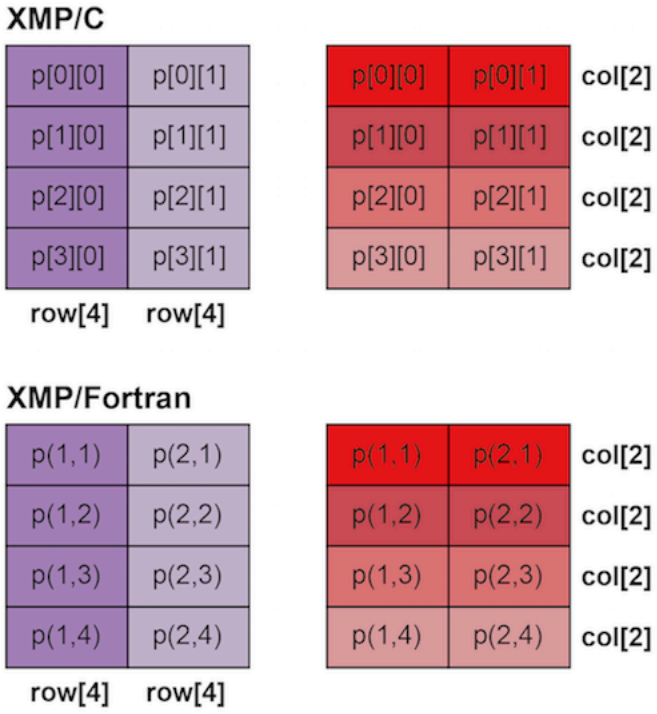


Fig. 5 Partial node set.

In XMP/C, both p[0][0] and p[0][1] will be row[0]. Likewise, p[0][0], p[1][0], p[2][0] and p[3][0] will be col[0] in each execution context. In XMP/Fortran, both p(1,1) and p(2,1) will be row(1). Likewise, p(1,1), p(1,2), p(1,3) and p(1,4) will be col(1) in each context.

Note: The syntactic meaning of asterisk symbols in the node set references are different when declaring a node set and regular expressions in on clauses.

2.2 `template` Directive

The template directive declares the name of a template and its shape. Templates are virtual arrays which used for data and work mapping. They can have multi-dimensional shapes.

2.2.1 One-dimensional Template

XcalableMP C
```
#pragma xmp template t[10]
```
XcalableMP Fortran
```
!$xmp template t(10)
```

The template directive declares 1-dimensional template t which has 10 elements. In XMP/C, each element has an unique index from t[0] to t[9]. Likewise, in XMP/-Fortran, the index starts from t(1) to t(10).

Hint: In general, the user declare templates which has the same size with the target data array where data/work mapping is done.

In XMP/Fortran, the start index of the template can be given by an arbitrary number to match the starting array index in the base language.

XcalableMP Fortran
```
!$xmp template t(-5:4)
```

The template directive declares 1-dimensional template t starting from t(-5) to t(4).

Note: In XMP/C, templates should start from 0 since array indices start from 0 in the C language.

2.2.2 Multi-dimensional Template

XcalableMP C
```
#pragma xmp template t[10][20]
```
XcalableMP Fortran
```
!$xmp template t(20,10)
```

The template directive declares 2-dimensional template t which has 10x20 elements. In XMP/C, the template has elements starting from t[0][0] to t[9][19]. Likewise, the template has elements starting from t(1,1) to t(20,10) in XMP/Fortran.

2.2.3 Dynamic Template

XcalableMP C
```
#pragma xmp template t[:]
```
XcalableMP Fortran
```
!$xmp template t(:)
```

A colon symbol is used instead of a number to declare 1-dimensional dynamic template t. The colon symbol indicates that the size of the template is undefined. The size of the template is determined at runtime by `template_fix` construct.

2.3 `distribute` Directive

The distribute directive specifies a distribution of the target template. The user can specify block, cyclic, block-cyclic, gblock (irregular data distribution) distribution, which can be chosen by the target application.

2.3.1 block distribution

XcalableMP C
```
#pragma xmp distribute t[block] onto p
```
XcalableMP Fortran
```
!$xmp distribute t(block) onto p
```

Target data is divided into contiguous blocks and distributed among nodes. When the size of the template is N and the number of nodes is K, the chunk size of each block will be ceil(N/K). For example, block distribution is useful for stencil computation which refers to neighborhood elements.

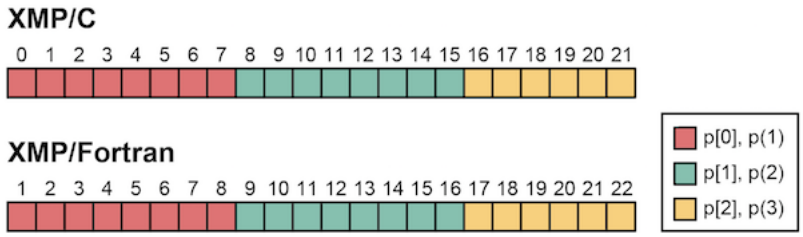
Note: Function ceil(x) returns the minimum integer value which is greater than x.

XcalableMP C
```
#pragma xmp nodes p[3]
#pragma xmp template t[22]
#pragma xmp distribute t[block] onto p
```
XcalableMP Fortran
```
!$xmp nodes p(3)
!$xmp template t(22)
!$xmp distribute t(block) onto p
```

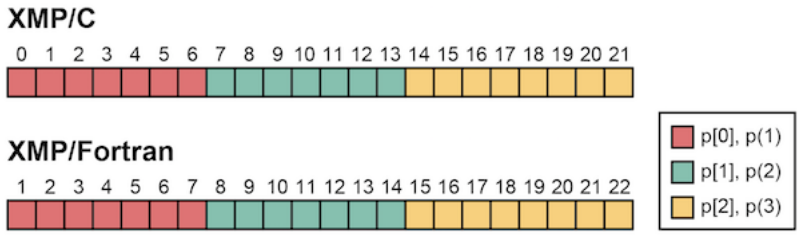
Since ceil(22/3) is 8, 8 elements will be allocated on p[0] and p[1]. And then, 6 elements will be allocated on p[2].

The user can specify the size of the block chunk explicitly. In that case, the remaining elements will be allocated on the last node.

XcalableMP C
```
#pragma xmp nodes p[3]
#pragma xmp template t[22]
#pragma xmp distribute t[block(7)] onto p
```



XcalableMP Fortran
```
!$xmp nodes p(3)
!$xmp template t(22)
!$xmp distribute t(block(7)) onto p
```



Seven elements will be allocated on the p[0] and p[1], as specified in the directive. And then remaining 8 elements will be allocated on the last node p[2].

2.3.2 Cyclic Distribution

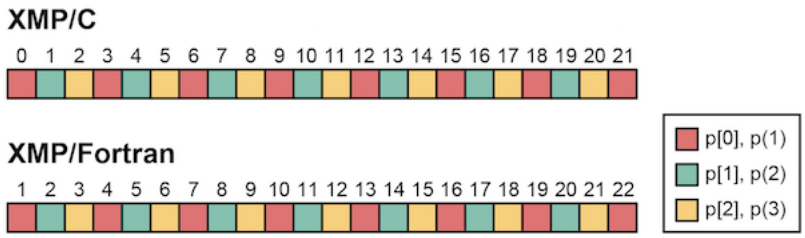
XcalableMP C
```
#pragma xmp distribute t[cyclic] onto p
```

XcalableMP Fortran
```
!$xmp distribute t(cyclic) onto p
```

Target data is divided into a chunk of a single element and distributed among nodes in a round-robin manner. Cyclic distribution is suitable for computation with an irregular load balance of data and computation.

XcalableMP C
```
#pragma xmp nodes p[3]
#pragma xmp template t[22]
#pragma xmp distribute t[cyclic] onto p
```

XcalableMP Fortran
```
!$xmp nodes p(3)
!$xmp template t(22)
!$xmp distribute t(cyclic) onto p
```



2.3.3 Block-cyclic Distribution

XcalableMP C
```
#pragma xmp distribute t[cyclic(w)] onto p
```

XcalableMP Fortran
```
!$xmp distribute t(cyclic(w)) onto p
```

Target data is divided into a contiguous block of size w and distributed among nodes in a round-robin manner. Block-cyclic distribution is suitable for computation which has an irregular load balance and references to neighborhood elements.

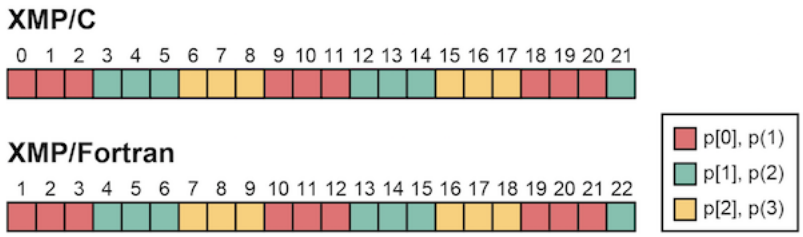
XcalableMP C
```
#pragma xmp nodes p[3]
#pragma xmp template t[22]
#pragma xmp distribute t[cyclic(3)] onto p
```

XcalableMP Fortran
```
!$xmp nodes p(3)
!$xmp template t(22)
!$xmp distribute t(cyclic(3)) onto p
```

2.3.4 Gblock Distribution

XcalableMP C
```
#pragma xmp distribute t[gblock(W)] onto p
```

XcalableMP Fortran
```
!$xmp distribute t(gblock(W)) onto p
```



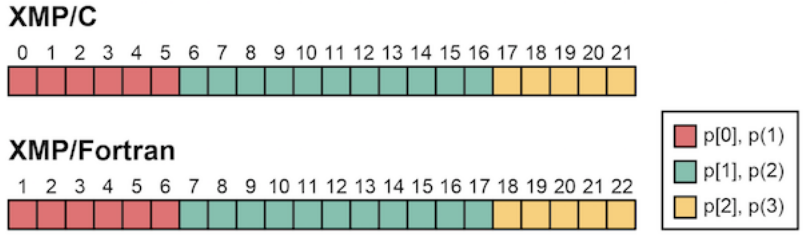
Array W is a mapping array which is used for irregular data distribution. W[k]/W(k) elements will be allocated on node p(k). The user can specify a special type of data distribution explicitly by using mapping arrays (e.g. distribution of triangular matrix).

XcalableMP C

```
#pragma xmp nodes p[3]
#pragma xmp template t[22]
int W[3] = {6, 11, 5};
#pragma xmp distribute t[gblock(W)] onto p
```

XcalableMP Fortran

```
!$xmp nodes p(3)
!$xmp template t(22)
integer, parameter :: W(3) = (/6,11,5/)
!$xmp distribute t(gblock(W)) onto p
```



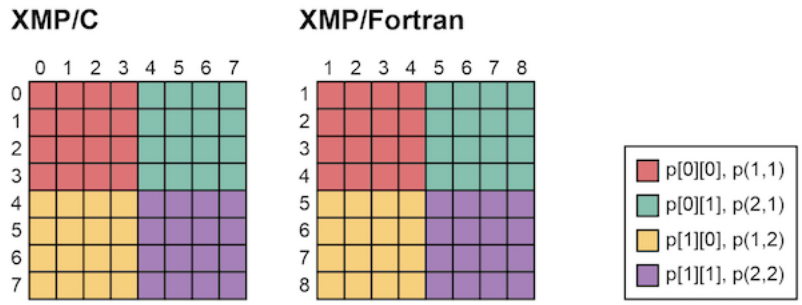
The user can specify an asterisk symbol instead of a mapping array in gblock. In that case, data distribution will be determined at runtime by using `template_fix` construct.

2.3.5 Distribution of Multi-dimensional Template

The user can distribute a multi-dimensional template with a (single-/multi-dimensional) node set.

XcalableMP C
```
#pragma xmp nodes p[2][2]
#pragma xmp template t[10][10]
#pragma xmp distribute t[block][block] onto p
```
XcalableMP Fortran
```
!$xmp nodes p(2,2)
!$xmp template t(10,10)
!$xmp distribute t(block,block) onto p
```

The distribute directive declares data distribution of a 2-dimensional template by using a 2-dimensional node set. Each dimension of the template is divided by block distribution on a node set p.

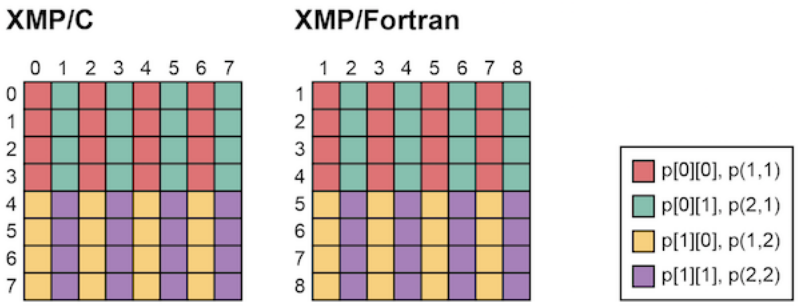


The user can specify a different distribution pattern to each dimension.

XcalableMP C
```
#pragma xmp nodes p[2][2]
#pragma xmp template t[10][10]
#pragma xmp distribute t[block][cyclic] onto p
```
XcalableMP Fortran
```
!$xmp nodes p(2,2)
!$xmp template t(10,10)
!$xmp distribute t(cyclic,block) onto p
```

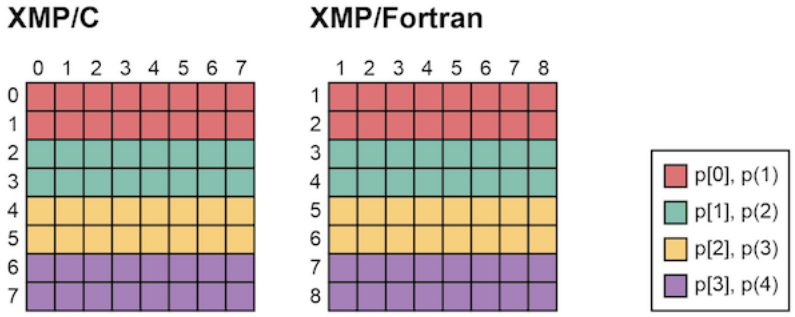
When an asterisk symbol is given in the distribute directive instead of distribution type, the target dimension will remain undistributed. In the following example, the first dimension will be distributed in a block manner and the second dimension will remain undistributed.

XcalableMP C
```
#pragma xmp nodes p[4]
#pragma xmp template t[10][10]
#pragma xmp distribute t[block][*] onto p
```



XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(10,10)
!$xmp distribute t(*,block) onto p
```



2.4 `align` Directive

The align directive performs data mapping and distributes data among nodes by using a distributed template. The align directive should be given after the target array definition.

2.4.1 Normal Alignment

XcalableMP C

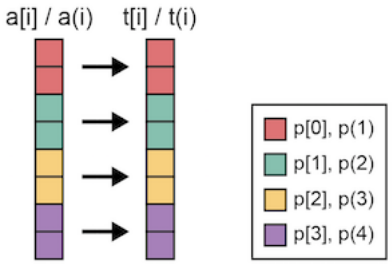
```
#pragma xmp nodes p[4]
#pragma xmp template t[8]
#pragma xmp distribute t[block] onto p
```

```
int a[8];
#pragma xmp align a[i] with t[i]
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(8)
!$xmp distribute t(block) onto p
integer :: a(8)
!$xmp align a(i) with t(i)
```

The align directive aligns the owner node of a[i] with t(i), a distributed template. As a result, array a is distributed among the node set p.



The align directive also can be used for multi-dimensional arrays.

XcalableMP C

```
#pragma xmp nodes p[2][2]
#pragma xmp template t[8][8]
#pragma xmp distribute t[block][block] onto p
int a[8][8];
#pragma xmp align a[i][j] with t[i][j]
```

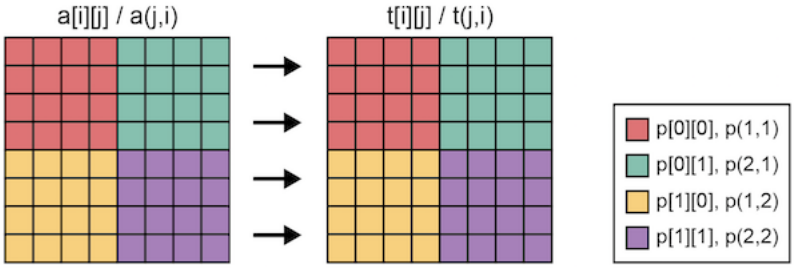
XcalableMP Fortran

```
!$xmp nodes p(2,2)
!$xmp template t(8,8)
!$xmp distribute t(block,block) onto p
integer :: a(8,8)
!$xmp align a(j,i) with t(j,i)
```

2.4.2 Special Alignment

Collapse

The user can align a 2-dimensional array with a 1-dimensional template.



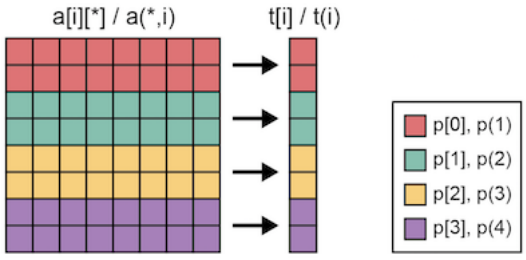
XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[8]
#pragma xmp distribute t[block] onto p
int a[8][8];
#pragma xmp align a[i][*] with t[i]
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(8)
!$xmp distribute t(block) onto p
integer :: a(8,8)
!$xmp align a(*,i) with t(i)
```

When an asterisk symbol is given in the array reference in the align directive, the specified dimension is not distributed among the node set. In the sample program, the first dimension of the array a is distributed among node set p while the second dimension is duplicated.



In XMP/C, a[0:2][:] will be allocated on p[0]. Likewise, a(:,1:2) will be allocated on p(1) in XMP/Fortran.

Replicate

The user also can align a 1-dimensional array with a multi-dimensional template.

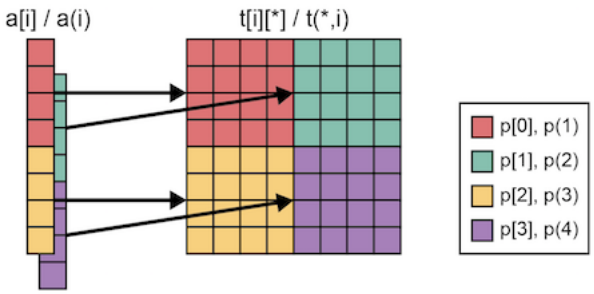
XcalableMP C

```
#pragma xmp nodes p[2][2]
#pragma xmp template t[8][8]
#pragma xmp distribute t[block][block] onto p
int a[8];
#pragma xmp align a[i] with t[i][*]
```

XcalableMP Fortran

```
!$xmp nodes p(2,2)
!$xmp template t(8,8)
!$xmp distribute t(block,block) onto p
integer :: a(8)
!$xmp align a(i) with t(*,i)
```

When an asterisk symbol is given in the template reference in the align directive, the owner nodes of the specified dimension will have duplicated elements of the target array.



In XMP/C, a[0:4] will be duplicated and allocated on p[0][0] and p[0][1]. Likewise, a(1:4) will be allocated on p(1,1) and p(2,1) in XMP/Fortran.

2.5 `template_fix` Construct

The `template_fix` construct defines the size and distribution of an unfixed template. It is also used when a distributed array is allocated at runtime.

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[:]
#pragma xmp distribute t[block] onto p
double *a;
#pragma xmp align a[i] with t[i]

int n = 100;
```

```
#pragma xmp template_fix t[n]
a = xmp_malloc(xmp_desc_of(a), n);
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(:)
!$xmp distribute t(block) onto p
real, allocatable :: a(:)
integer :: n
!$xmp align a(i) with t(i)

n = 100
!$xmp template_fix t(n)
allocate(a(n))
```

First, declare a template the size of which is undefined using the ”:” notation. Second, align a pointer in XMP/C and an allocatable array in XMP/Fortran with the template. Third, fix the size of the template with a `template_fix` construct. Finally, allocate the array with the `xmp_malloc()` builtin function in XMP/C and the allocate statement in XMP/Fortran.

Note: `template_fix` constructs can be applied to a template only once.

The `template_fix` construct can also be used to define a mapping array of a template that is distributed in “gblock(*)” at declaration.

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[:]
#pragma xmp distribute t[gblock(*)] onto p
double *a;
#pragma xmp align a[i] with t[i]

int n = 100;
int m[] = {40,30,20,10};

#pragma xmp template_fix[gblock(m)] t[n]
a = xmp_malloc(xmp_desc_of(a), n);
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(:)
!$xmp distribute t(gblock) onto p
real, allocatable :: a(:)
integer :: n, m(4)
!$xmp align a(i) with t(i)

n = 100
```

```
m(:) = (/40,30,20,10/)
!$xmp template_fix(gblock(m)) t(n)
allocate(a(n))
```

3 Work Mapping

3.1 `task/tasks` Construct

A task construct generates a task, which is executed by the specified nodes. A tasks construct asserts that surrounding task constructs can be executed in parallel.

This page shows some examples of the task construct other than those in Tutorial (Global-view).

3.1.1 `task` Construct

The on clause of the task construct specifies the node set that executes the task.

XcalableMP C

```
#include <stdio.h>
#pragma xmp nodes p[4]

int main(){
  int num = xmpc_node_num();
#pragma xmp task on p[1:3]
{
  printf("%d: Hello\n", num);
}

  return 0;
}
```

XcalableMP Fortran

```
program main
!$xmp nodes p(4)
  integer :: num

  num = xmp_node_num()
!$xmp task on p(2:4)
  write(*,*) num, ": Hello"
!$xmp end task

end program main
```

In the above example, nodes p[1], p[2], and p[3] execute the printf() function and p[1] outputs “1: Hello” in XMP/C, and p(2), p(3), and p(4) execute the write statement, and p(2) outputs “2: Hello” in XMP/Fortran.

Note that a new node set is generated by each task construct. Consider inserting a bcast construct into a task.

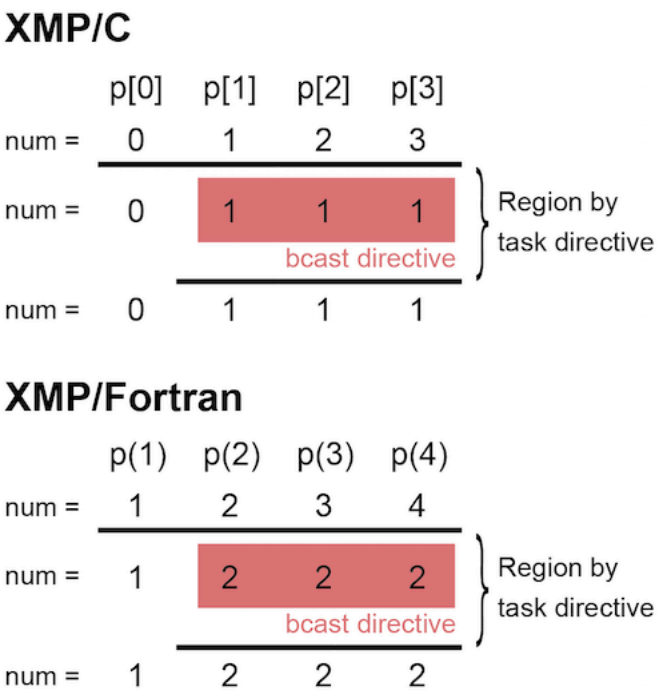
XcalableMP C

```
#pragma xmp task on p[1:3]
{
#pragma xmp bcast (num)
}
```

XcalableMP Fortran

```
!$xmp task on p(2:4)
!$xmp bcast (num)
!$xmp end task
```

This bcast construct is executed by the node set specified by the task construct. Thus, the node p[1] broadcasts the value to p[2] and p[3] in XMP/C, and p(2) to p(3) and p(4) in XMP/Fortran.



The bcast construct in the above code is equivalent to that in the following code, where it is executed by a new node set that is explicitly declared.

XcalableMP C
```
#pragma xmp nodes q[3] = p[1:3]
#pragma xmp bcast (num) on q
```
XcalableMP Fortran
```
!$xmp nodes q(3) = p(2:4)
!$xmp bcast (num) on q
```

Note that the task is executed by the node set specified by the on clause. Therefore, xmpc_node_num() and xmp_node_num() return the id the node set.

For example, consider inserting xmpc_node_num() or xmp_node_num() into the task in the first program.

XcalableMP C
```
#include <stdio.h>
#pragma xmp nodes p[4]

int main(){
#pragma xmp task on p[1:3]
{
  printf("%d: Hello\n", xmpc_node_num());
}

  return 0;
}
```
XcalableMP Fortran
```
program main
!$xmp nodes p(4)

!$xmp task on p(2:4)
  write(*,*) xmp_node_num(), ": Hello"
!$xmp end task

end program main
```

The node p[1] outputs "0: Hello" in XMP/C and p(2) "1: Hello" in XMP/Fortran.

Note: A new node set should be collectively generated by all of the executing nodes at the point of task construct unless it is surrounded by a tasks construct. In the above example, p[0] in XMP/C and p(1) in XMP/Fortran must execute the task construct.

3.1.2 `tasks` Construct

Consider that each of two tasks invokes a function.

XcalableMP C

```
#pragma xmp nodes p[4]

#pragma xmp task on p[0:2]
{
  func_a();
}
#pragma xmp task on p[2:2]
{
  func_b();
}
```

XcalableMP Fortran

```
!$xmp nodes p(4)

!$xmp task on p(1:2)
  call func_a()
!$xmp end task
!$xmp task on p(3:4)
  call func_b()
!$xmp end task
```

In the above example, the two tasks cannot be executed in parallel because those on clauses must be evaluated by all of the executing nodes.

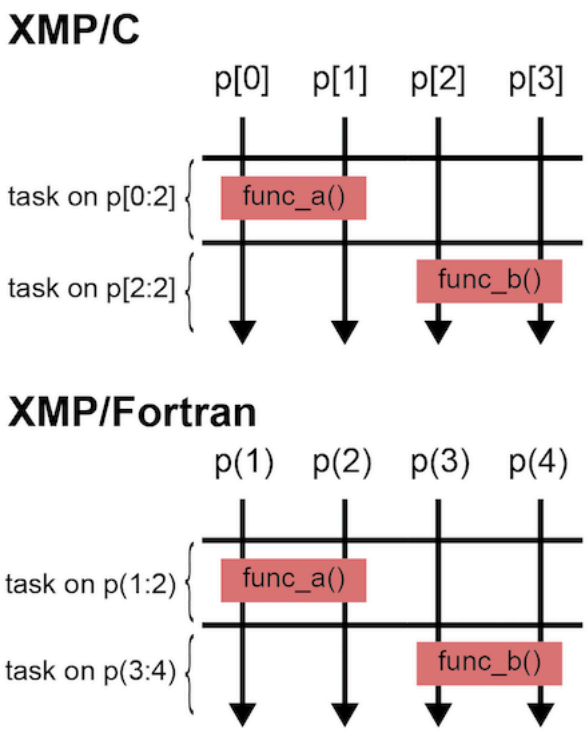
Use the tasks construct to execute multiple tasks in parallel.

XcalableMP C

```
#pragma xmp nodes p[4]

#pragma xmp tasks
{
#pragma xmp task on p[0:2]
{
  func_a();
}
#pragma xmp task on p[2:2]
{
  func_b();
}
}
```

XcalableMP Fortran

```
!$xmp nodes p(4)

!$xmp tasks
!$xmp task on p(1:2)
  call func_a()
!$xmp end task
```



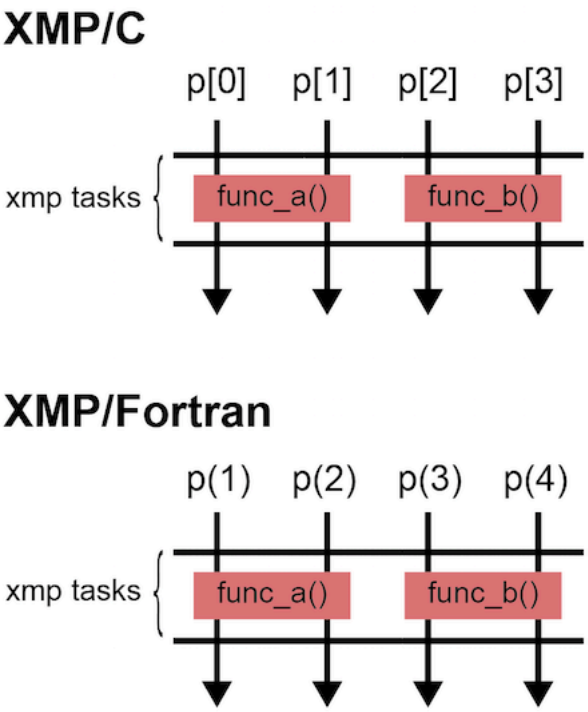

```
!$xmp task on p(3:4)
  call func_b()
!$xmp end task
!$xmp end tasks
```

The node sets specified by the on clauses of task constructs surrounded by a task construct can be executed in parallel.

3.2 `loop` Construct

The loop directive is used to parallelize a loop. Distributed arrays in such a loop must fulfill the following two conditions:

There is no data/control dependence among the iterations. In other words, the iterations of the loop can be executed in any order to produce the same result. An element of a distributed array is accessed only by the node that owns the element.



3.2.1 Accessing Distributed Array

The programs below are examples of a right loop directive and a loop statement. The condition 1. is satisfied because i is the only one index of the distributed array a that is accessed within the loop, and the condition 2 is also satisfied because the indices of the template in the on clause of the loop directive is identical to that of the distributed array.

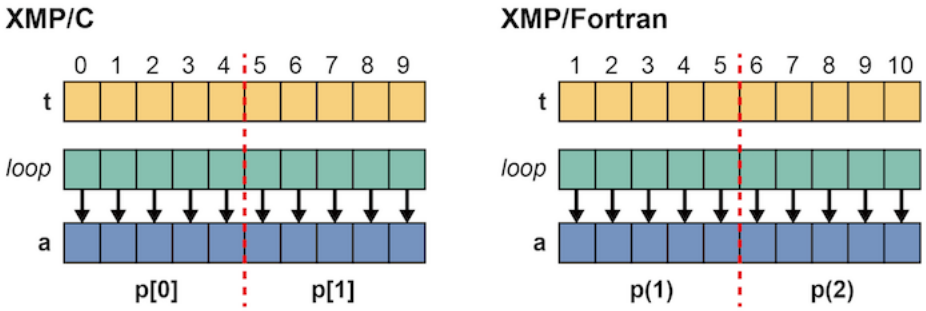
XcalableMP C

```
#pragma xmp nodes p[2]
#pragma xmp template t[10]
#pragma xmp distribute t[block] onto p

int main(){
  int a[10];
#pragma xmp align a[i] with t[i]

#pragma xmp loop on t[i]
  for(int i=0;i<10;i++)
    a[i] = i;
```

```
  return 0;
}
```

XcalableMP Fortran

```
program main
!$xmp nodes p(2)
!$xmp template t(10)
!$xmp distribute t(block) onto p
  integer a(10)
!$xmp align a(i) with t(i)

!$xmp loop on t(i)
  do i=1, 10
    a(i) = i
  enddo

end program main
```



Is it possible to parallelize the below loops whose bounds are shrunk?

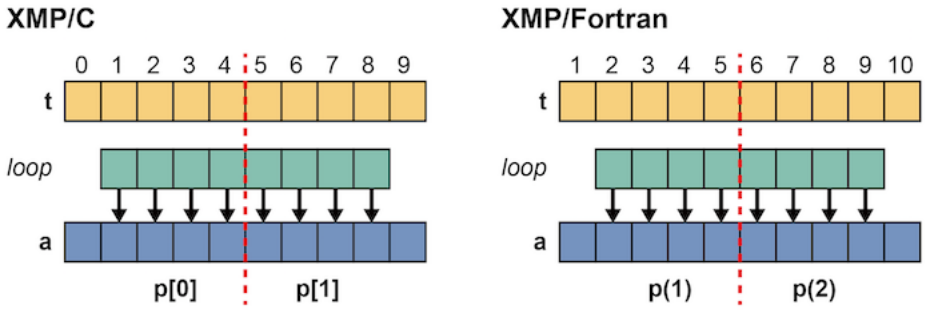
XcalableMP C

```
#pragma xmp loop on t[i]
  for(int i=1;i<9;i++)
    a[i] = i;
```

XcalableMP Fortran

```
!$xmp loop on t(i)
  do i=2, 9
    a(i) = i
  enddo
```

In this case, the conditions 1 and 2 are satisfied and therefore it is possible to parallelize them. In XMP/C, p[0] processes the indices from one to four and p[1] from five to eight. In XMP/Fortran, p(1) processes the indices from two to five and p(2) from six to nine.



Next, is it possible to parallelize the below loops in which the index of the distributed array is different?

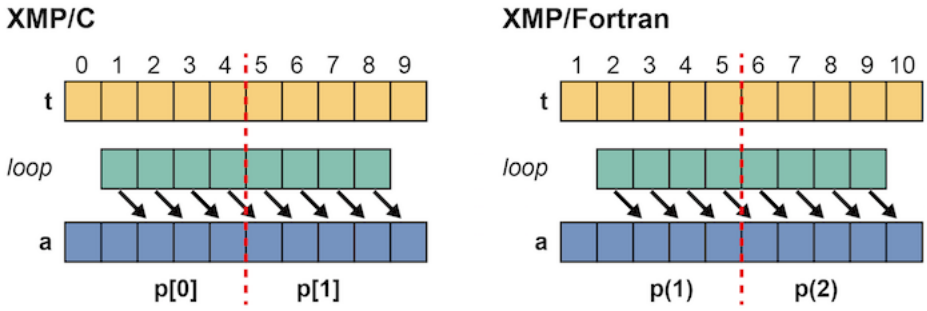
XcalableMP C

```
#pragma xmp loop on t[i]
  for(int i=1;i<9;i++)
    a[i+1] = i;
```

XcalableMP Fortran

```
!$xmp loop on t(i)
  do i=2, 9
    a(i+1) = i
  enddo
```

In this case, the condition 1 is satisfied but 2 is not, and therefore it is not possible to parallelize them. In XMP/C, p[0] tries to access a[5] but does not own it. In XMP/Fortran, p(1) tries to access a(6) but does not own it.



3.2.2 Reduction Computation

The serial programs below are examples of the reduction computation.

C

```c
#include <stdio.h>

int main(){
  int a[10], sum = 0;

  for(int i=0;i<10;i++){
    a[i] = i+1;
    sum += a[i];
  }

  printf("%d\n", sum);

  return 0;
}
```

Fortran

```
program main
  integer :: a(10), sum = 0

  do i=1, 10
    a(i) = i
    sum = sum + a(i)
  enddo

  write(*,*) sum

end program main
```

If the above loops are parallelized only with the loop directive, the value of the variable sum varies from node to node because it is calculated on each node.

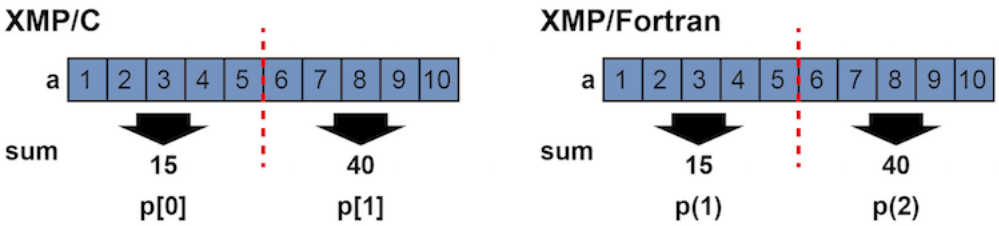
XcalableMP C

```c
#pragma xmp loop on t[i]
   for(int i=0;i<10;i++){
     a[i] = i+1;
     sum += a[i];
   }
```

XcalableMP Fortran

```
!$xmp loop on t(i)
  do i=1, 10
    a(i) = i
    sum = sum + a(i)
  enddo
```

Then, add the reduction clause to the loop directive.



XcalableMP C

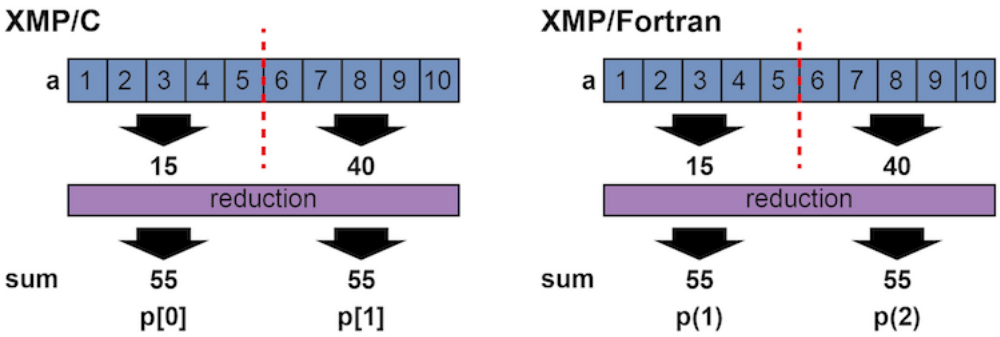
```
#include <stdio.h>
#pragma xmp nodes p[2]
#pragma xmp template t[10]
#pragma xmp distribute t[block] onto p

int main(){
  int a[10], sum = 0;
#pragma xmp align a[i] with t[i]

#pragma xmp loop on t[i] reduction(+:sum)
  for(int i=0;i<10;i++){
    a[i] = i+1;
    sum += a[i];
  }

  printf("%d\n", sum);

  return 0;
}
```

XcalableMP Fortran

```
program main
!$xmp nodes p(2)
!$xmp template t(10)
!$xmp distribute t(block) onto p
  integer :: a(10), sum = 0
!$xmp align a(i) with t(i)

!$xmp loop on t(i) reduction(+:sum)
  do i=1, 10
    a(i) = i
    sum = sum + a(i)
  enddo

  write(*,*) sum
```

```
end program main
```

An operator and target variables for reduction are specified in a reduction clause. In the above examples, a “+” operator is specified for the reduction computation to produce a total sum among nodes.



Operations that can be used in a reduction computation are limited to the following associative ones.

C

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

Fortran

```
+
*
-
.and.
.or.
.eqv.
.neqv.
```

```
max
min
iand
ior
ieor
firstmax
firstmin
lastmax
lastmin
```

Note: If the reduction variable is a type of floating point, the difference of the order of the executions can make a little bit difference between serial and parallel executions

3.2.3 Parallelizing Nested Loop

Parallelization of nested loops can be specified similarly for a single loop.

XcalableMP C

```
#pragma xmp nodes p[2][2]
#pragma xmp template t[10][10]
#pragma xmp distribute t[block][block] onto p

int main(){
  int a[10][10];
#pragma xmp align a[i][j] with t[i][j]

#pragma xmp loop on t[i][j]
  for(int i=0;i<10;i++)
    for(int j=0;j<10;j++)
      a[i][j] = i*10+j;

  return 0;
}
```

XcalableMP Fortran

```
program main
!$xmp nodes p(2,2)
!$xmp template t(10,10)
!$xmp distribute t(block,block) onto p
  integer :: a(10,10)
!$xmp align a(j,i) with t(j,i)

!$xmp loop on t(j,i)
  do i=1, 10
```

```
    do j=1, 10
      a(j,i) = i*10+j
    enddo
  enddo

end program main
```

3.3 `array` Construct

The array construct is for work mapping of array assignment statements.

XcalableMP C

```
#pragma xmp align a[i] with t[i]
  :
#pragma xmp array on t[0:N]
a[0:N] = 1.0;
```

XcalableMP Fortran

```
!$xmp align a(i) with t(i)
  :
!$xmp array on t(1:N)
a(1:N) = 1.0
```

The above is equivalent to the below.

XcalableMP C

```
#pragma xmp align a[i] with t[i]
  :
#pragma xmp loop on t[i]
for(int i=0;i<N;i++)
  a[i] = 1.0;
```

XcalableMP Fortran

```
!$xmp align a(i) with t(i)
  :
!$xmp loop on t(i)
do i=1, N
  a(i) = 1.0
enddo
```

This construct can also be applied to multi-dimensional arrays. The triplet notation enables specifying operations for all elements of the array.

XcalableMP C

```
#pragma xmp align a[i][j] with t[i][j]
  :
#pragma xmp array on t[:][:]
a[:][:] = 1.0;
```

XcalableMP Fortran
```
!$xmp align a(j,i) with t(j,i)
  :
!$xmp array on t(:,:)
a(:,:) = 1.0
```

Note: The template appearing in the on clause must have the same shape of arrays in the following statement. The right-hand side value must be identical among all nodes because the array construct is a collective operation.

4 Data Communication

4.1 shadow Directive / reflect Construct

The stencil computation frequently appears in scientific computations, where array elements a[i-1] and a[i+1] are referenced to update a[i]. If a[i] is on the boundary area of a distributed array on a node, a[i+1] may reside on another node.

Because it costs largely to copy a[i+1] from the neighboring node to update each a[i], a technique of copying collectively elements on the neighboring node to the area added to the distributed array on each node is usually adopted. In XMP, such additional region is called “shadow.”

4.1.1 Declare Shadow

Widths of lower/upper bounds are the same

Shadow areas can be declared with the shadow directive. In the example below, an array a has shadow areas of size one on both the lower and upper bounds.

XcalableMP C
```
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t[block] onto p
double a[16];
#pragma xmp align a[i] with t[i]
#pragma xmp shadow a[1]
```

XcalableMP Fortran
```
!$xmp nodes p(4)
!$xmp template t(16)
!$xmp distribute t(block) onto p
real :: a(16)
```

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

In the figure above, colored elements are those that each node owns and white ones are shadow.

Note: Distributed arrays in a cyclic manner cannot have shadow.

Widths of lower/upper bounds are different

For some programs, it is natural that the widths of the shadow area on the lower and upper bounds are different. There is also a case where the shadow area exists only on either of the bounds. In the example below, it is declared that a distributed array a has a shadow area of width one only on the upper bound.

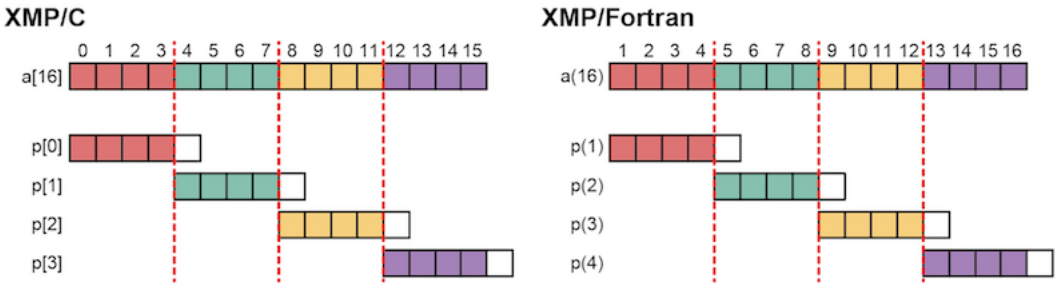
XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t(block) onto p
double a[16];
#pragma xmp align a[i] with t[i]
#pragma xmp shadow a[0:1]
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(16)
!$xmp distribute t(block) onto p
real :: a(16)
!$xmp align a(i) with t(i)
!$xmp shadow a(0:1)
```

The values on the left- and right-hand sides of a colon designate the widths on the lower and upper bounds, respectively.



4.1.2 Update Shadow

General

To copy data to shadow areas from neighboring nodes, use the reflect directive. In the example below, an array a having shadow areas of width one on each the upper and lower bounds is reflected.

XcalableMP C

```
#pragma xmp reflect (a)

#pragma xmp loop on t[i]
for(int i=1;i<15;i++)
  a[i] = (a[i-1] + a[i] + a[i+1])/3;
```

XcalableMP Fortran

```
!$xmp reflect (a)

!xmp loop on t(i)
do i=2, 15
  a(i) = (a(i-1) + a(i) + a(i+1))/3
enddo
```

With this reflect directive, in XMP/C, node p[1] sends an element a[4] to the shadow area on the upper bound on node p[0] and a[7] to the shadow area on the lower bound on p[2]; p[0] sends an element a[3] to the shadow area on the lower bound on p[1], and p[2] sends a[8] to the shadow area on the upper bound on p[1].

Similarly, in XMP/Fortran, node p(2) sends an element a(5) to the shadow area on the upper bound on node p(1) and a(8) to the shadow area on the lower bound on p(3); p(1) sends an element a(4) to the shadow area on the lower bound on p(2), and p(3) sends a(9) to the shadow area on the upper bound on p(2).

Specify Width

The default behavior of a reflect directive is to update the whole of the shadow area declared by a shadow directive. However, there are some cases where a specific part of the shadow area is to be updated to reduce the communication size in a point of the code.

To update only a specific part of the shadow area, add the width clause to the reflect directive.

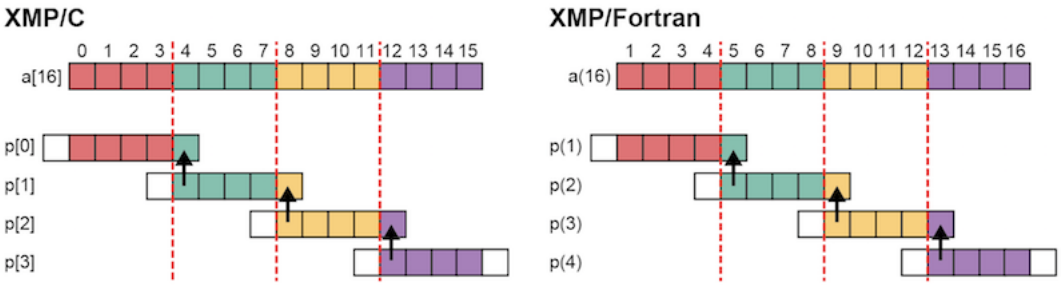
The values on the left- and right-hand sides of a colon in the width clause designate the widths on the lower and upper bounds to be updated, respectively. In the example below, only the shadow area on the upper bound is updated.

XcalableMP C

```
#pragma xmp reflect (a) width(0:1)
```

XcalableMP Fortran

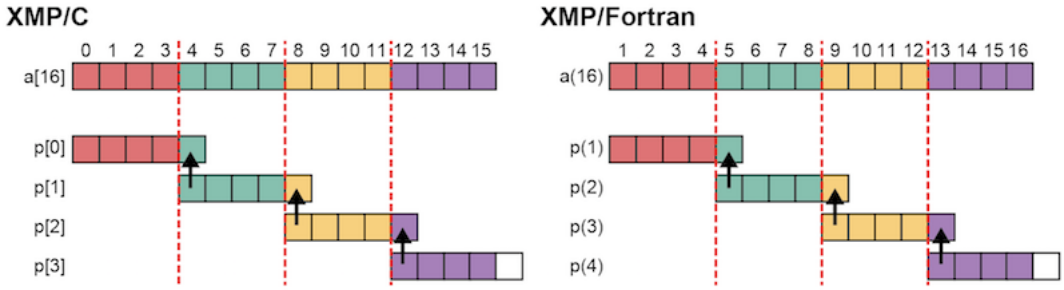
```
!$xmp reflect (a) width(0:1)
```



Note: If the widths of the shadow areas to be updated on the upper and lower bounds are equal, that is, for example, width(1:1), you can abbreviate it as width(1).

Note: It is not possible to update the shadow area on a particular node.

If no shadow area is specified on the lower bound, the reflect directive does not update it with or without a width clause. The below figure illustrates the behavior of a reflect directive for a distributed array a having a shadow area of width one only on the upper bound.



Update periodic shadow

The reflect directive does not update either the shadow area on the lower bound on the leading node or that on the upper bound on the last node. However, the values in such areas are needed for stencil computation if the computation needs a periodic boundary condition.

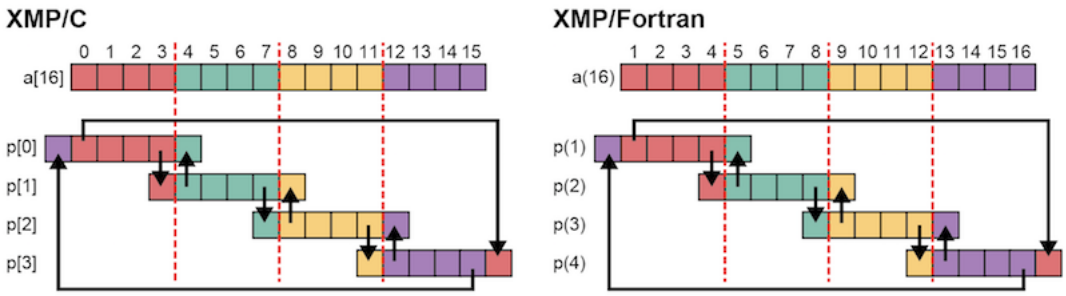
To update such areas, add a periodic qualifier into a width clause. Let ' s look at the following example where an array a having shadow areas of width one on both the lower and upper bounds appears.

XcalableMP C

```
#pragma xmp reflect (a) width(/periodic/1:1)
```

XcalableMP Fortran

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



The periodic qualifier has the following effects, in addition to that of a normal reflect directive: in XMP/C, node p[0] sends an element a[0] to the shadow area on the upper bound on node p[3], and p[3] sends a[15] to the shadow area on the lower bound on p[0]; in XMP/Fortran, node p(1) sends an element a(1) to the shadow area on the upper bound on node p(4), and p(4) sends a(16) to the shadow area on the lower bound on p(1).

Note: If the widths of the shadow areas to be updated on the upper and lower bounds are equal, as shown by width(/periodic/1:1) in the above example, you can abbreviate it as width(/periodic/1).

4.1.3 Multi-dimensional Shadow

The shadow directive and reflect construct can be applied to distributed arrays multiple dimensions. The following programs are the examples for two-dimensional distribution.

XcalableMP C

```
#pragma xmp nodes p[3][3]
#pragma xmp template t[9][9]
#pragma xmp distribute t[block][block] onto p
double a[9][9];
#pragma xmp align a[i][j] with t[i][j]
#pragma xmp shadow a[1][1]
   :
#pragma xmp reflect (a)
```

XcalableMP Fortran

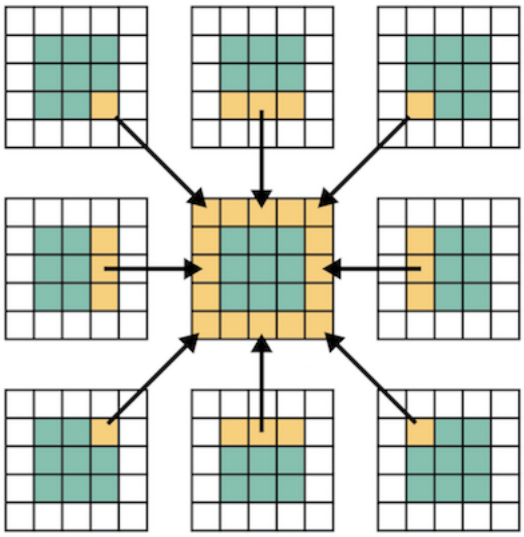
```
!$xmp nodes p(3,3)
!$xmp template t(9,9)
!$xmp distribute t(block,block) onto p
real :: a(9,9)
!$xmp align a(j,i) with t(j,i)
!$xmp shadow a(1,1)
   :
!$xmp reflect (a)
```

The central node receives the shadow data from the surrounding eight nodes. The shadow areas of the other nodes are also updated, which is omitted in the figure.

For some applications, data from ordinal directions are not necessary. In such a case, the data communication from/to the ordinal directions can be avoided by adding an orthogonal clause to a reflect construct.

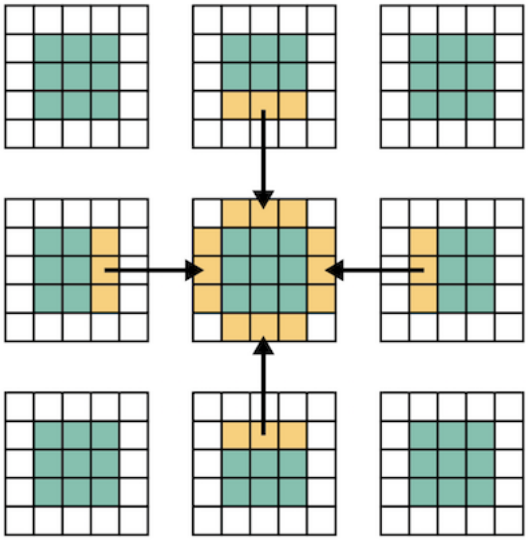
XcalableMP C

```
#pragma xmp reflect (a) orthogonal
```



XcalableMP Fortran

```
!$xmp reflect (a) orthogonal
```



Note: The orthogonal clause is effective only for arrays more than one dimension of which is distributed.

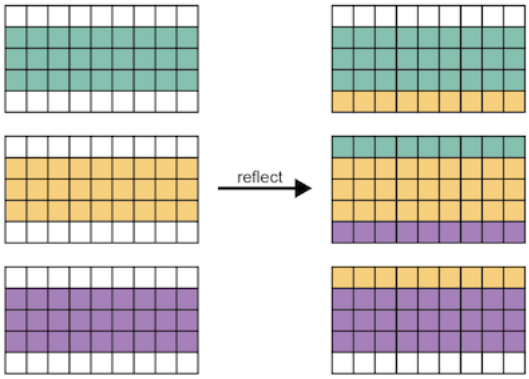
Besides, you can also add shadow areas to only specified dimension.

XcalableMP C

```
#pragma xmp nodes p[3]
#pragma xmp template t[9]
#pragma xmp distribute t[block] onto p
double a[9][9];
#pragma xmp align a[i][*] with t[i]
#pragma xmp shadow a[1][0]
  :
#pragma xmp reflect (a)
```

XcalableMP Fortran

```
!$xmp nodes p[3]
!$xmp template t[9]
!$xmp distribute t[block] onto p
real :: a(9,9)
!$xmp align a(*,i) with t(i)
!$xmp shadow a(0,1)
  :
!$xmp reflect (a)
```



In the array specified in the shadow directive, 0 is set as the shadow width in dimensions which are not distributed.

4.2 `gmove` Construct

You can describe a communication for distributed arrays in the form of assignment statements by using gmove directive.

There are three modes in gmove; “collective mode”, “in mode,” and “out mode.” While collective mode executes two-sided communication among the executing nodes, in/out modes execute one-sided communication among tasks with a task directive. While in mode uses get communication, out mode uses put communication.

4.2.1 Collective Mode

Distributed array

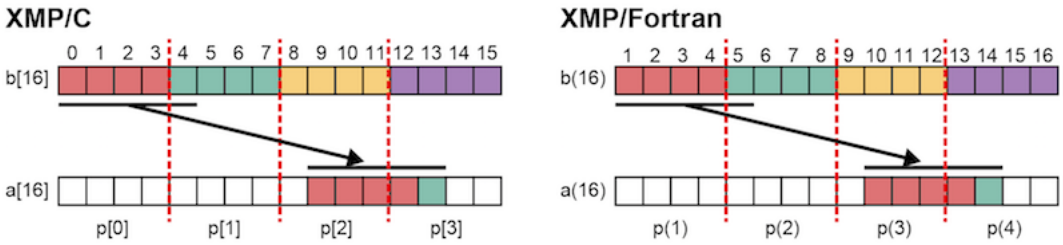
Copying a part of array a to array b. Array assignment statements in a gmove construct uses triplet.

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t[block] onto p
int a[16], b[16];
#pragma xmp align a[i] with t[i]
#pragma xmp align b[i] with t[i]
     :
#pragma xmp gmove
  a[9:5] = b[0:5];
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(16)
!$xmp distribute t(block) onto p
integer :: a(16), b(16)
!$xmp align a(i) with t(i)
!$xmp align b(i) with t(i)
     :
!$xmp gmove
  a(10:14) = b(1:5)
```



In XMP/C, p[0] sends b[0] - b[3] to p[2] - p[3], and p[1] sends b[4] to p[3]. Similarly, in XMP/Fortran, p(1) sends b(1) - b(4) to p(3) - p(4), and p(2) sends b(5) to p(4).

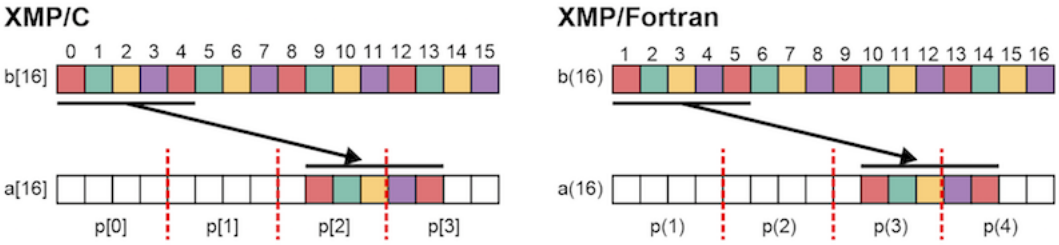
In this example, it is assignment statements between distributed arrays with the same shape. XMP also supports to assign it with the different shape.

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t1[16]
#pragma xmp template t2[16]
#pragma xmp distribute t1[cyclic] onto p
#pragma xmp distribute t2[block] onto p
int a[16], b[16];
#pragma xmp align a[i] with t1[i]
#pragma xmp align b[i] with t2[i]
     :
#pragma xmp gmove
  a[9:5] = b[0:5];
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t1(16)
!$xmp template t2(16)
!$xmp distribute t1(cyclic) onto p
!$xmp distribute t2(block) onto p
integer :: a(16), b(16)
!$xmp align a(i) with t1(i)
!$xmp align b(i) with t2(i)
     :
!$xmp gmove
  a(10:14) = b(1:5)
```



While array a is distributed in a cyclic manner, array b is distributed in a block manner.

In XMP/C, p[0] sends b[0] and b[4] to p[2] and p[3]. p[1] sends b[1] to p[2]. Each element of p[2] and p[3] will be copied locally. Similarly, in XMP/Fortran,

p(1) sends b(1) and b(5) to p(3) and p(4). p(2) sends b(2) to p(3). Each element of p(3) and p(4) will be copied locally.

Note: If the number of elements specified on the right-hand side is other than 1, it will not work properly if the number of elements differs between the right-hand side and the left-hand side.

By using this method, the shape of a distributed array can be changed during calculation.

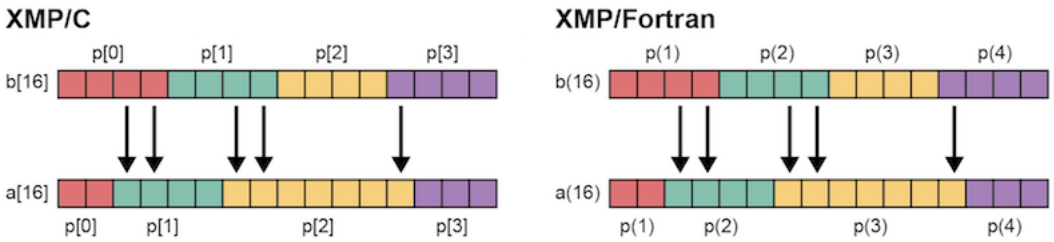
XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t1[16]
#pragma xmp template t2[16]
int W[4] = {2,4,8,2};
#pragma xmp distribute t1[gblock(W)] onto p
#pragma xmp distribute t2[block] onto p
int a[16], b[16];
#pragma xmp align a[i] with t1[i]
#pragma xmp align b[i] with t2[i]
     :
#pragma xmp gmove
  a[:] = b[:];
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t1(16)
!$xmp template t2(16)
integer :: W(4) = (/2,4,7,3/)
!$xmp distribute t1(gblock(W)) onto p
!$xmp distribute t2(block) onto p
integer :: a(16), b(16)
!$xmp align a(i) with t1(i)
!$xmp align b(i) with t2(i)
     :
!$xmp gmove
  a(:) = b(:)
```

In this example, copying all elements of array b which is distributed in a block manner to array a which is distributed in a gblock manner. In arrays a and b, communication occurs only for elements whose responsible nodes do not match (the arrow means communication between nodes in figures).



Scalar

In an assignment statement, if one element is specified on the right-hand side and plural elements are specified on the left-hand side, the operation will be broadcast communication.

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t[block] onto p
int a[16], b[16];
#pragma xmp align a[i] with t[i]
#pragma xmp align b[i] with t[i]
     :
#pragma xmp gmove
  a[9:5] = b[0];
```

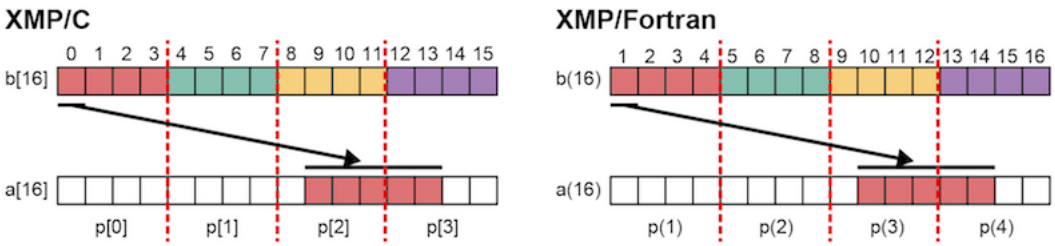
XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(16)
!$xmp distribute t(block) onto p
integer :: a(16), b(16)
!$xmp align a(i) with t(i)
!$xmp align b(i) with t(i)
     :
!$xmp gmove
  a(10:14) = b(1)
```

In this example, in XMP/C, an element array b[0] of node p[0] will be broadcasted to the specified index of node p[2] and p[3]. Similarly, in XMP/Fortran, an element array b(1) of node p(1) will be broadcasted to the specified index of node p(3) and p(4).

Duplicated array and scalar

Not only distributed arrays but also duplicated arrays and scalar variables can be described on the right-hand side.



XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t[block] onto p
int a[16], b[16], c;
#pragma xmp align a[i] with t[i]
     :
#pragma xmp gmove
   a[9:5] = b[0:5];

#pragma xmp gmove
   a[9:5] = c;
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(16)
!$xmp distribute t(block) onto p
integer :: a(16), b(16), c
!$xmp align a(i) with t(i)
     :
!$xmp gmove
   a(10:14) = b(1:5)

!$xmp gmove
   a(10:14) = c
```

In this example, duplicated array and scalar variable are copied to distributed array locally. For this reason, communication does not occur.

Distributed array with different dimension

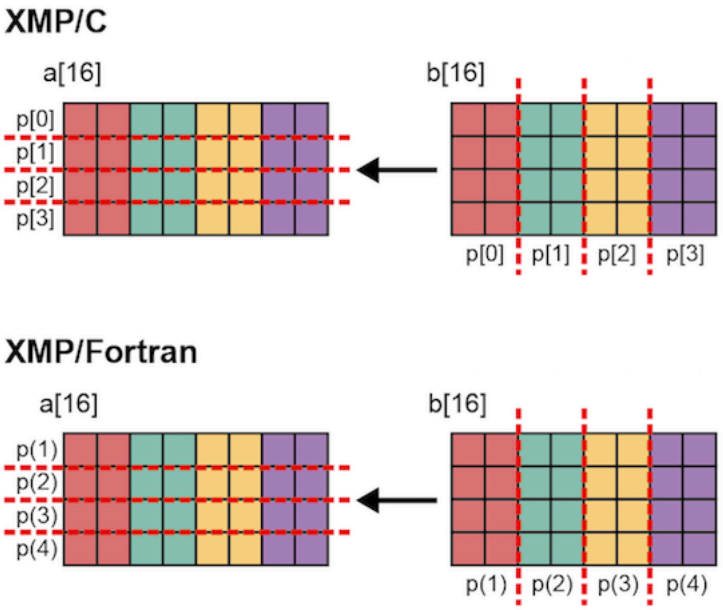
XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t1[8]
#pragma xmp template t2[16]
#pragma xmp distribute t1[block] onto p
```

```
#pragma xmp distribute t2[block] onto p
int a[8][16], b[8][16];
#pragma xmp align a[i][*] with t1[i]
#pragma xmp align b[*][i] with t2[i]
     :
#pragma xmp gmove
  a[0][:] = b[0][:];
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t1(8)
!$xmp template t2(16)
!$xmp distribute t1(block) onto p
!$xmp distribute t2(block) onto p
integer :: a(16,8), b(8,16)
!$xmp align a(*,i) with t1(i)
!$xmp align b(i,*) with t2(i)
     :
#pragma xmp gmove
  a(:,1) = b(:,1)
```



In this example, in XMP/C, b[0][0:2] of p[0], b[0][2:2] of p[1], b[0][4:2] of p[2] and b[0][6:2] of p[3] are copied to a[0][:] of p[0]. Similarly, in XMP/Fortran, b(1:2,1)

of p(1), b(3:4,1) of p(2), b(5:6,1) of p(3) and b(7:8,1) of p(4) are copied to a(:,1) of p(1).

4.2.2 In Mode

It operates as in mode by setting in clause to gmove directive.

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[4]
#pragma xmp distribute t[block] onto p
double a[4], b[4];
#pragma xmp align a[i] with t[i]
#pragma xmp align b[i] with t[i]
   :
#pragma xmp task on p[0:2]
#pragma xmp gmove in
  a[0:2] = b[2:2]
#pragma xmp end task
```

XcalableMP Fortran

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

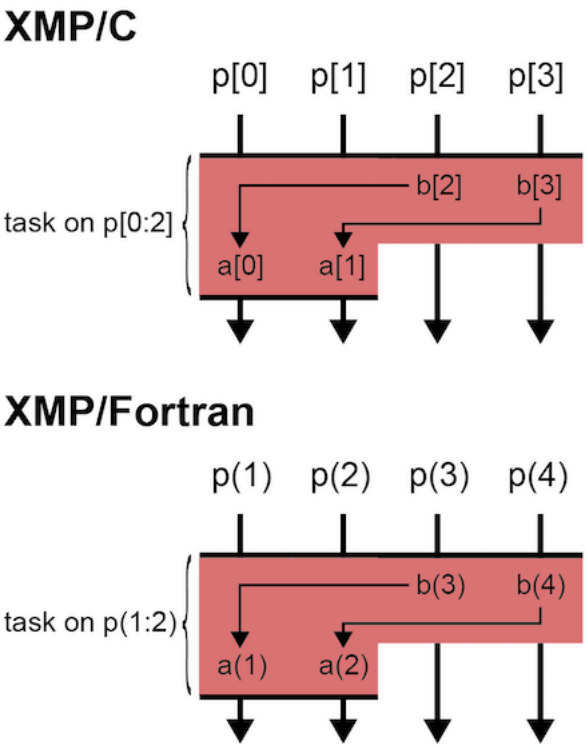
In this example, the task directive divides the node set of 4 nodes into two nodes, the first-half and the second-half. In gmove directive which is in mode, it executes get communication from array of second-half node to array of first-half node.

4.2.3 Out Mode

It operates as out mode by setting out clause to gmove directive

XcalableMP C

```
#pragma xmp nodes p[4]
#pragma xmp template t[4]
#pragma xmp distribute t[block] onto p
double a[4], b[4];
```

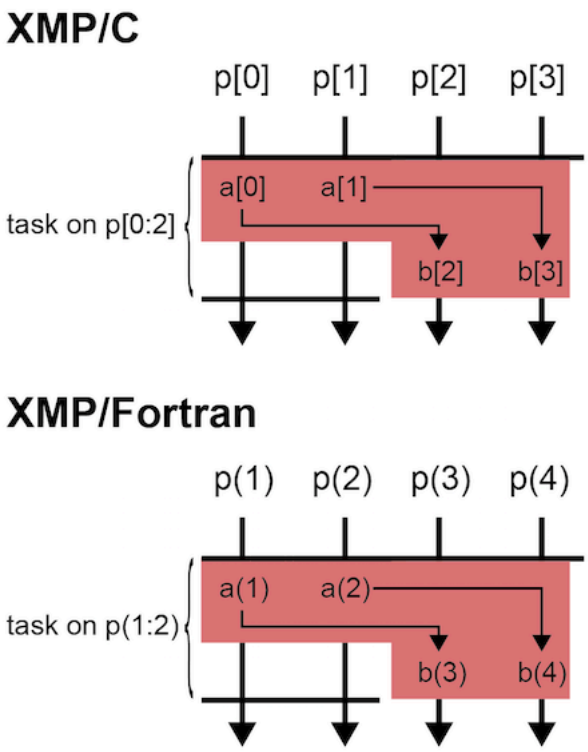



```
#pragma xmp align a[i] with t[i]
#pragma xmp align b[i] with t[i]
   :
#pragma xmp task on p[0:2]
#pragma xmp gmove out
  b[2:2] = a[0:2]
#pragma xmp end task
```

XcalableMP Fortran

```
!$xmp nodes p(4)
!$xmp template t(4)
!$xmp distribute t(block) onto p
real :: a(4), b(4)
!$xmp align a(i) with t(i)
!$xmp align b(i) with t(i)
   :
!$xmp task on p(1:2)
!$xmp gmove out
  b(3:4) = a(1:2)
!$xmp end task
```

In this example, it just reversed the assignment statement of the in mode. In gmove directive which is out mode, it executes put communication from array of first-half node to array of second-half node.



4.3 barrier Construct

Execute barrier synchronization.

XcalableMP C

```
#pragma xmp barrier
```

XcalableMP Fortran

```
!$xmp barrier
```

You can set the barrier range by using the on clause. In the below example, barrier synchronization occurs only in the first two nodes of p.

XcalableMP C

```
#pragma xmp barrier on p[0:2]
```

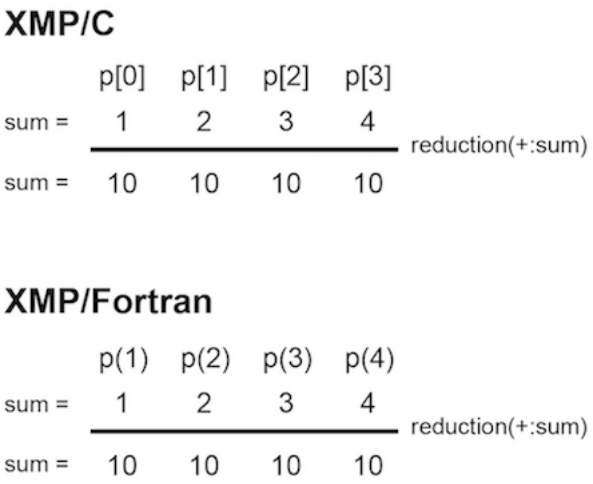
XcalableMP Fortran
```
!$xmp barrier on p(1:2)
```

4.4 reduction Construct

Execute reduction operation. It has the same meaning as the reduction clause of loop construct, but the reduction directive can be described anywhere.

XcalableMP C
```
#pragma xmp nodes p[4]
  :
sum = xmpc_node_num() + 1;
#pragma xmp reduction (+:sum)
```

XcalableMP Fortran
```
!$xmp nodes p(4)
  :
sum = xmp_node_num()
!$xmp reduction (+:sum)
```



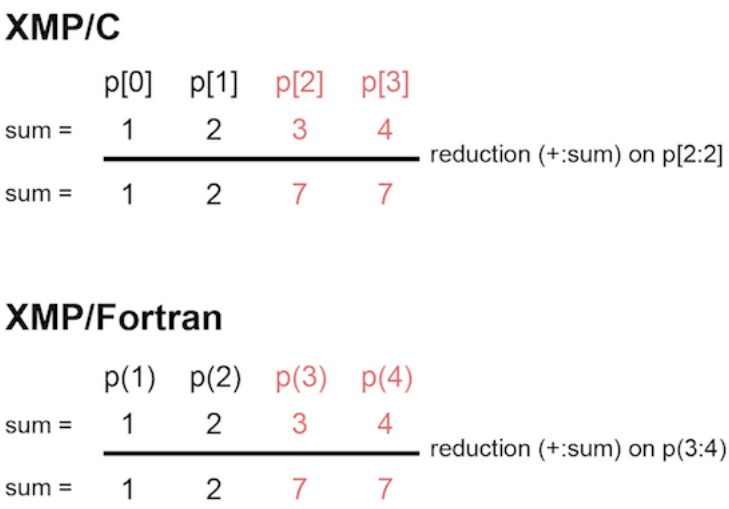
You can set the range of the node set by using the on clause. In the below example, only the values of the last two nodes in four nodes are targets to reduction operation.

XcalableMP C
```
#pragma xmp nodes p[4]
  :
```

```
sum = xmpc_node_num() + 1;
#pragma xmp reduction (+:sum) on p[2:2]
```

XcalableMP Fortran

```
!$xmp nodes p(4)
  :
 sum = xmp_node_num()
 !$xmp reduction (+:sum) on p(3:4)
```



The specifiable operators are as follows.

XcalableMP C

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

XcalableMP Fortran

```
+
*
-
.and.
.or.
```

```
.eqv.
.neqv.
max
min
iand
ior
ieor
```

Note: Since the reduction clause needs a loop statement, operators of firstmax, firstmin, lastmax, and lastmin are required. But, since the reduction directive does not need a loop statement, there are no such operators.

Note: Similar to the reduction clause, the reduction directive may have slightly different results from sequential execution and parallel execution, because of depending on the calculation order when the reduction variable is a floating-point type.

4.5 bcast Construct

The bcast directive broadcasts variables which are held by the node, specified in the from clause, to the node set specified in the on clause. If there is no from clause, the first node of the target node set is the starting point. If there is no on clause, the current set of nodes will be covered.

In the below example, the first node of the node set p is the starting point.

XcalableMP C

```
#pragma xmp nodes p[4]
  :
num = xmpc_node_num() + 1;
#pragma xmp bcast (num)
```

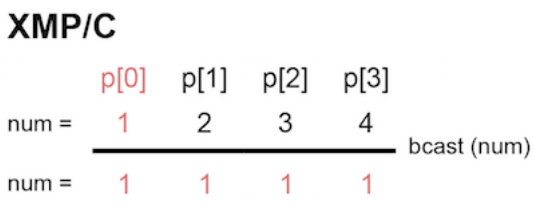
XcalableMP Fortran

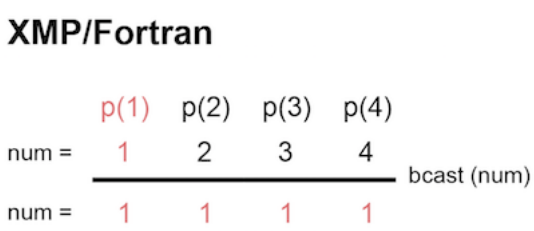
```
!$xmp nodes p(4)
  :
num = xmp_node_num()
!$xmp bcast (num)
```

In the below example, starting from the last node of the node set p by the from clause.

XcalableMP C

```
#pragma xmp nodes p[4]
  :
```

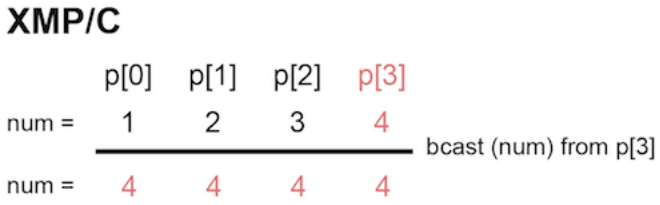


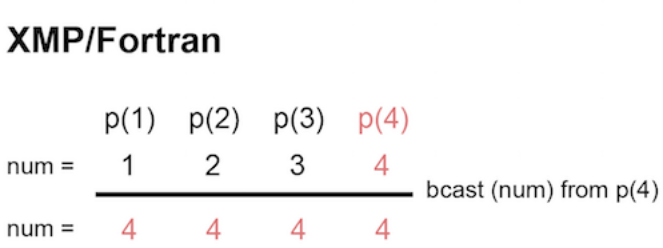



```
num = xmpc_node_num() + 1;
#pragma xmp bcast (num) from p[3]
```

XcalableMP Fortran

```
!$xmp nodes p(4)
  :
num = xmp_node_num()
!$xmp bcast (num) from p(4)
```





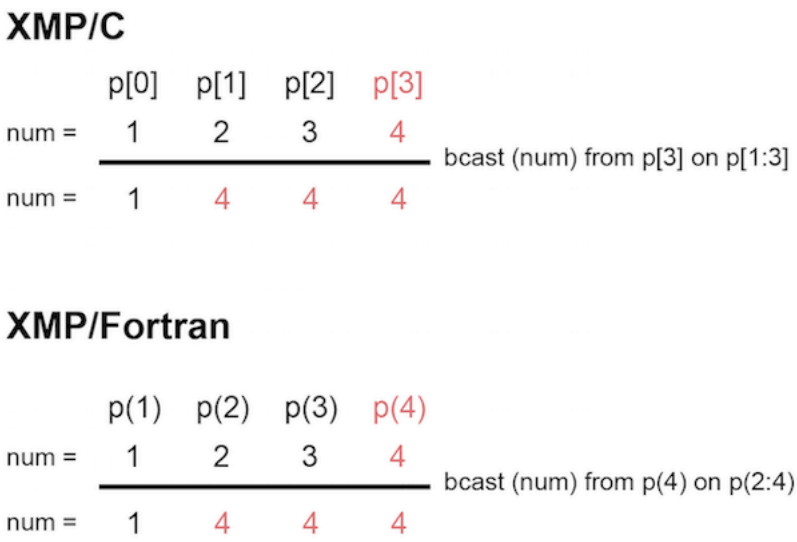
In the below example, only the values of the last three nodes in 4 nodes are targets for communication by the on clause.

XcalableMP C

```
#pragma xmp nodes p[4]
  :
sum = xmpc_node_num() + 1;
#pragma xmp bcast (num) from p[3] on p[1:3]
```

XcalableMP Fortran

```
!$xmp nodes p(4)
  :
 sum = xmp_node_num()
 !$xmp bcast (num) from p(4) on p(2:4)
```



4.6 wait_async Construct

Communication directives (reflect, gmove, reduction, bcast, reduce_shadow) can execute asynchronous communication by attaching the "async" clause. The wait_async directive is used to guarantee the completion of their asynchronous communication.

XcalableMP C

```
#pragma xmp bcast (num) async(1)
    :
#pragma xmp wait_async (1)
```

XcalableMP Fortran
```
!$xmp bcast (num) async(1)
        :
!$xmp wait_async (1)
```

Since the bcast directive has an async clause, communication may not be completed immediately after the bcast directive. Completion of that communication is guaranteed with the wait_async directive whose the same value as the async clause is specified. Therefore, between the bcast directive and the wait_async directive, you can not operate on the variable specified in the bcast directive.

Hint: By performing calculations without a dependency relationship with the variable specified by the bcast directive after the bcast directive, overlap of communication and calculation can be performed, so the total calculation time may be small.

Note: Values that can be specified for async clause are int type in XMP/C, and integer type in XMP/Fortran.

4.7 reduce_shadow Construct

The reduce_shadow directive adds the value of the shadow to the value of the source element.

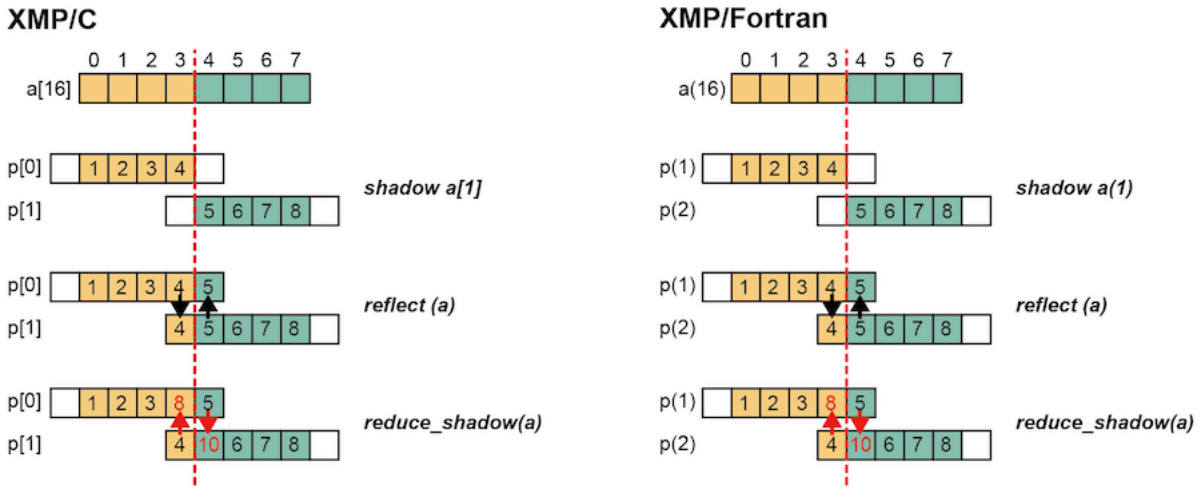
XcalableMP C
```
#pragma xmp nodes p[2]
#pragma xmp template t[8]
#pragma xmp distribute t[block] onto p
int a[8];
#pragma xmp align a[i] with t[i]
#pragma xmp shadow a[1]
 :
#pragma xmp loop on t[i]
  for(int i=0;i<8;i++)
    a[i] = i+1;

#pragma xmp reflect (a)
#pragma xmp reduce_shadow (a)
```

XcalableMP Fortran
```
!$xmp nodes p(2)
!$xmp template t(8)
!$xmp distribute t(block) onto p
```

```
  integer a(8)
!$xmp align a(i) with t(i)
!$xmp shadow a(1)

!$xmp loop on t(i)
  do i=1, 8
    a(i) = i
  enddo

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

The shadow directive adds one shadow to the distributed array a of each node. Next, the reflect directive will update shadow between neighborhood nodes. Finally, the reduce_shadow directive adds the value of the shadow to the value of the source element.

In XMP/C, a[3] of p[0] has a value of 8, and a[4] of p[1] has a value of 10. Similarly, in XMP/Fortran, a(4) of p(1) has a value of 8, and a(5) of p(2) has a value of 10.



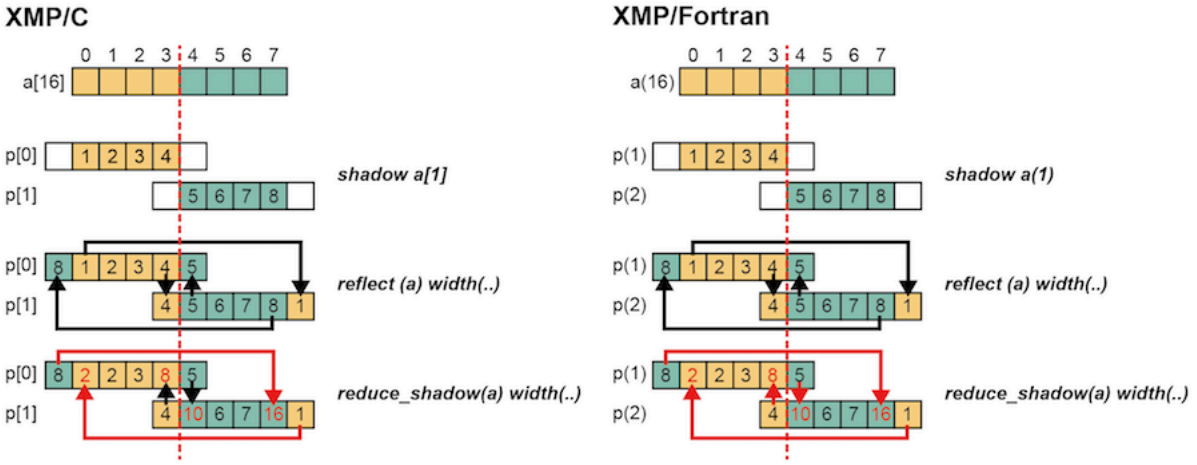
You can add the periodic modifier to the width clause to execute periodic region updates.

XcalableMP C

```
#pragma xmp reflect (a) width(/periodic/1)
#pragma xmp reduce_shadow (a) width(/periodic/1)
```

XcalableMP Fortran

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



In addition to the first example, in XMP/C, a[0] of p[0] has a value of 2, and a[7] of p[1] has a value of 16. Similarly, in XMP/Fortran, a(1) in p(1) has a value of 2, and a(8) in p(2) has a value of 16.

5 Local-view Programming

5.1 Introduction

The user uses Coarray in the local-view model to describe one-sided communication. In XMP, put/get communication and some synchronization functions are supported.

If the target system supports Remote Direct Memory Access（RDMA） in the hardware, one-sided communication in the local-view model can achieve better performance compared to the global-view model. However, it requires more effort to describe parallel program since all communication should be specified in detail.

XMP/Fortran supports Coarray in the standard Fortran 2008. Coarray in XMP/C has its own original syntax since the C language does not support Coarray.

Note: XMP/Fortran has upward compatibility with Fortran 2008.

The basic unit of execution in the local-view is called “image” while it is called “node” in the global-view model. The two words have the same meaning in XMP.

5.2 Coarray Declaration

XcalableMP C

```
int a[10]:[*];
```

XcalableMP Fortran

```
integer a(10)[*]
```

In XMP/C, the user declares a Coarray by adding :[] (Coarray dimension) after the array declaration. In XMP/Fortran, the user declares a Coarray by adding [] after the array declaration. The asterisk symbol is used in both languages.

Note: Based on Fortran 2008, the Coarray should have the same size of the entire execution node group.

Coarray can be accessed from other images using assignment statements. Of course, coarray can be also accessed from your image like ordinary array.

5.3 One-sided Communication

5.3.1 Put Communication

When the Coarray reference appears in the left-hand side in an assignment statement, it causes put communication.

XcalableMP C

```
int a[10]:[*], b[10];

if (xmpc_this_image() == 0)
  a[0:3]:[1] = b[3:3];
```

XcalableMP Fortran

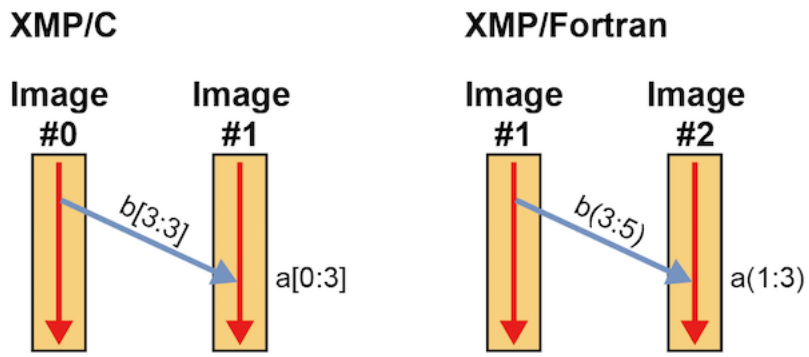
```
integer a(10)[*]
integer b(10)

if (this_image() == 1) then
  a(1:3)[2] = b(3:5)
end if
```

The integer number in the Coarray dimension specifies the targer image. Each image index starts with 0 in XMP/C and starts with 1 in XMP/Fortran. `xmpc_this_image()` in XMP/C and `this_image()` XMP/Fortran return the current image index.

Note: In XMP/Fortran, image index starts with 1 while it uses [] (similar to C style for array dimension) to specify Coarray dimension based on the standard Fortran 2008.

Note: When Coarray dimension appears on both side, 3 nodes (target, source, current node) involve the communication.

In the above example, XMP/C puts b[3:3] on image 0 to a[0:3] on image 1. XMP/Fortran puts b(3:5) on image 1 to a(1:3) on image 2. The following figure illustrates the one-sided communication done by Corray.



Note: The directives in the global-view model invoke point-to-point communication. On the other hand, Coarrays in the local-view model invoke one-sided communication.

5.3.2 Get Communication

When a Coarray appears in the right-hand side in the assignment statement, it causes get communication.

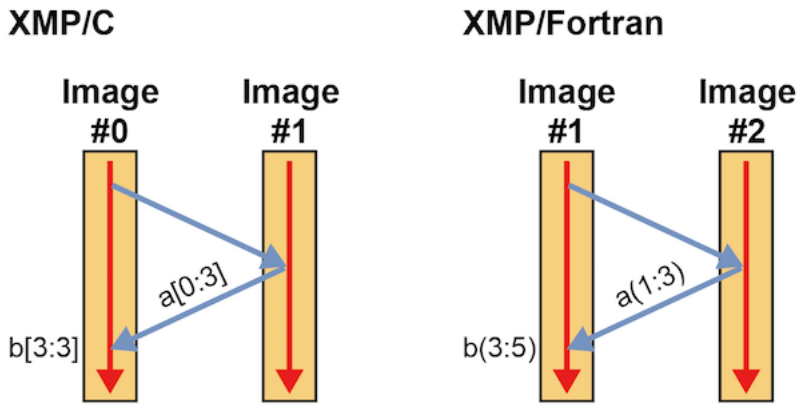
XcalableMP C

```
int a[10]:[*], b[10];

if (xmpc_this_image() == 0)
  b[3:3] = a[0:3]:[1];
```

XcalableMP Fortran

```
integer a(10)[*]
integer b(10)

if (this_image() == 1) then
  b(3:5) = a(1:3)[2]
end if
```

In the above program, XMP/C gets a[0:3] from image 1 and stores them on b[3:3] of image 0. XMP/Fortran gets a(1:3) from image 2 and stores them on b(3:5) of image 1. The following figure illustrates Coarray get communication.



Hint

As illustrated get needs an extra step to send a request to the target node. Put communication achieves better performance than get since there is no such extra step.

5.3.3 Synchronization

Here, we introduce “sync all” which is most frequently used among Coarray synchronization functions.

XcalableMP C

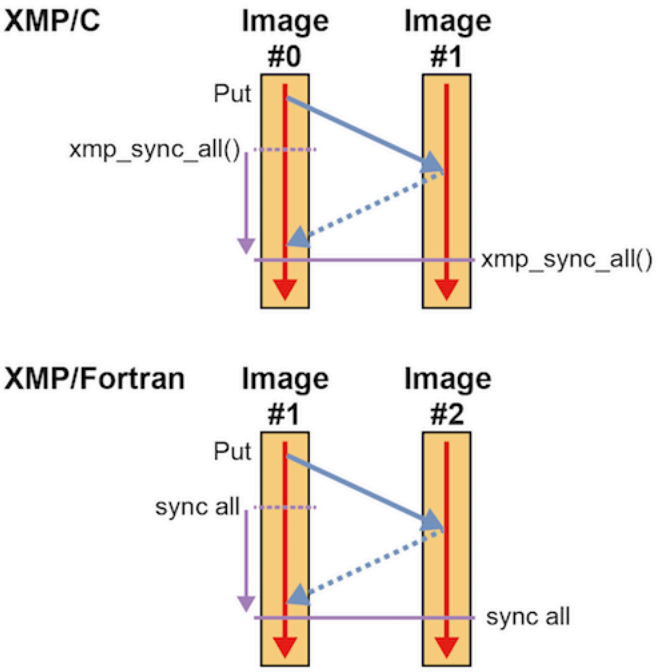
```
void xmp_sync_all(int *status)
```

XcalableMP Fortran

```
sync all
```

The “sync all” waits all issued one-sided communication and invokes barrier synchronization among the entire images.

In the above example, the left image puts data to the right image and both nodes invoke sync all. When both nodes finish sync all, the execution continues after the synchronization point.



5.4 Synchronization Statements

5.4.1 sync all

XcalableMP C

```
void xmp_sync_all(int *status)
```

XcalableMP Fortran

```
sync all
```

Barrier synchronization is performed among all images after completing all one side communications. For details, see Tutorial (Local-view).

5.4.2 sync images

XcalableMP C

```
void xmp_sync_images(int num, int *image-set, int *status)
```

XcalableMP Fortran

```
sync images (image-set)
```

Barrier synchronization is performed among the specified images after completing all one side communications.

XcalableMP C
```
int image_set[3] = {0,1,2};
xmp_sync_images(3, image_set, NULL);
```
XcalableMP Fortran
```
integer :: image_set(3) = (/ 1, 2, 3/)
sync images (image_set)
```

5.4.3 sync memory

XcalableMP C
```
void xmp_sync_memory(int *status)
```
XcalableMP Fortran
```
sync memory
```

Wait for completion of all one side communications. This function does not include barrier synchronization unlike sync all and sync images, so it is executed only locally.

5.5 `post/wait` Construct

5.6 `lock/unlock` Construct

6 Procedure Interface

Procedure calls in XMP are the same as the base language. Procedure calls between different languages and external library are also possible if the base language supports them.

In the below example, sub1() calls sub2() with a distributed array as an argument.

XcalableMP C
```
void sub1(){
#pragma xmp nodes p[2]
#pragma xmp template t[10]
#pragma xmp distribute t[block] onto p
  double x[10];
#pragma xmp align x[i] with t[i]
  sub2(x);
}

void sub2(double a[10]){
#pragma xmp nodes p[2]
#pragma xmp template t[10]
```

```
#pragma xmp distribute t[block] onto p
  double a[10];
#pragma xmp align a[i] with t[i]
  :
}
```

XcalableMP Fortran

```
subroutine sub1()
!$xmp nodes p(2)
!$xmp template t(10)
!$xmp distribute t(block) onto p
  real x(10)
!$xmp align x(i) with t(i)
  call sub2(x)
end subroutine

subroutine sub2(a)
!$xmp nodes p(2)
!$xmp template t(10)
!$xmp distribute t(block) onto p
  real a(10)
!$xmp align a(i) with t(i)
  :
end subroutine
```

If you want to use distributed arrays in arguments as distributed arrays in the called procedure, you need to redefine the shape of the distributed array in the procedure.

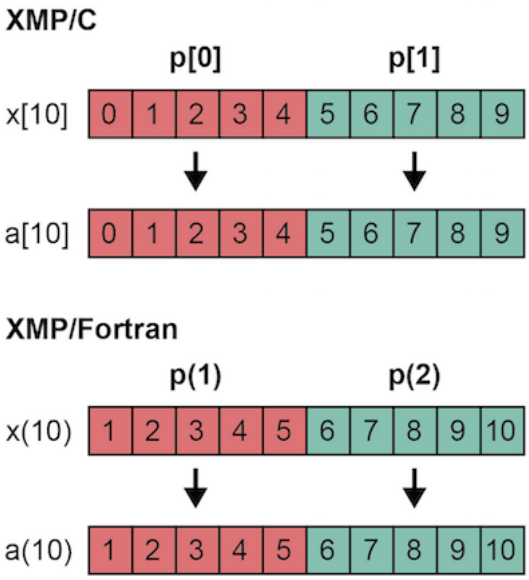
But, if you want to use the distributed array in the argument as a duplicate array in the called procedure, you do not need to redefine them.

XcalableMP C

```
void sub1(){
#pragma xmp nodes p[2]
#pragma xmp template t[10]
#pragma xmp distribute t[block] onto p
  double x[10];
#pragma xmp align x[i] with t[i]
  sub2(x);
}

void sub2(double a[5]){
  :
}
```

XcalableMP Fortran

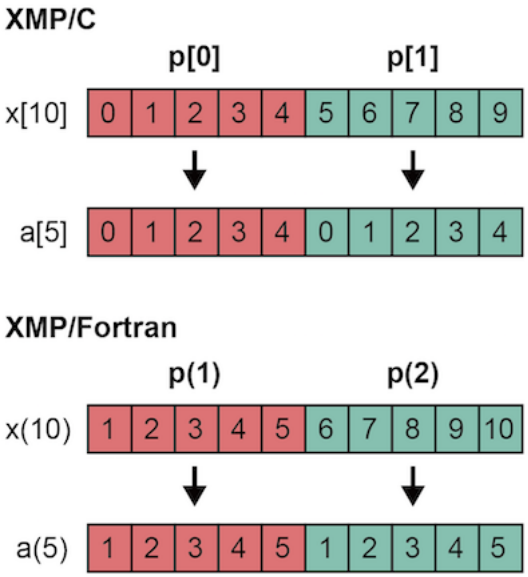
```
subroutine sub1()
!$xmp nodes p(2)
!$xmp template t(10)
```




```
!$xmp distribute t(block) onto p
  real x(10)
!$xmp align x(i) with t(i)
  call sub2(x)
end subroutine

subroutine sub2(a)
  real a(5)
  :
end subroutine
```

References

1. Robert W. Numrich and John Reid, “Co-Array Fortran for parallel programming”, ACM SIGPLAN Fortran Forum, Vol. 17, No. 2 (1998).
2. UPC Consortium, “UPC Specifications, v1.2”, Lawrence Berkeley National Lab (LBNL-59208) (2005).
3. David Callahan, Bradford L. Chamberlain and Hans P. Zima, “The Cascade High Productivity Language”, Proc. 9th Int’l. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004), pp. 52–60 (2004).


XMP/C
p[0]
p[1]
x[10]
0 1 2 3 4 5 6 7 8 9
a[5]
0 1 2 3 4 0 1 2 3 4
XMP/Fortran
p(1)
p(2)
x(10)
1 2 3 4 5 6 7 8 9 10
a(5)
1 2 3 4 5 1 2 3 4 5