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-view 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, programming with MPI is difficult because programmers must describe inter-process communications with consideration of the execution flow of their 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, programmers can execute their serial programs in parallel by inserting minimal directives into them. If the programmers specify data distribution with HPF directives, the compilers do all other tasks for parallelization (e.g. communication generation and work distribution). However, HPF was not widely accepted eventually because the compilers' automatic processing prevents

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

the programmers from performance tuning, and the performance depends heavily on the environment (e.g. compiler 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 new 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, the programmers specify explicitly the details of parallel programs on the basis of compiler directives to make their execution easy-to-understand. In particular, they can specify explicitly communication, synchronization, data mapping, and work mapping 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 programmers to implement parallel algorithms more easily.

In this chapter, an overview of the programming model and language specification of XMP is shown. You can find the latest and complete language specification of XMP in: XcalableMP Specification Working Group, XcalableMP Specification Version 1.4, `http://xcalablemp.org/download/spec/xmp-spec-1.4.pdf` (2018).

1.1 Target Hardware

The target of XcalableMP is distributed-memory multicomputers (Fig. 1). Each compute 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. via inter-node communication). However, it is assumed that accessing remote memory may be much slower than accessing local memory.

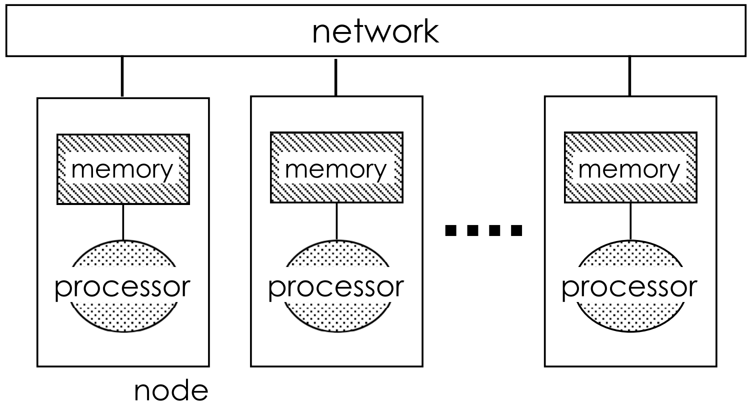

Fig. 1 Target hardware of XMP.

1.2 Execution Model

The execution entities in an XMP program are referred to as XMP **nodes** or, more simply, **nodes**, which has its own memory and can communicate with each other.

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) until it encounters an XcalableMP construct (Fig. 2).

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 (explicitly or implicitly) 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 the other **nodes**, so that each iteration of the loop or element of the assignment is independently executed by the **node** in which the specified data element resides.

When a **node** encounters a synchronization or a communication directive, synchronization or communication occurs between it and the 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 specified.

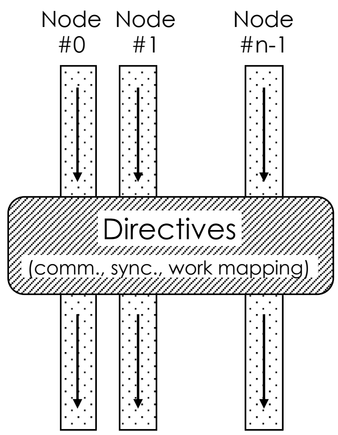

Fig. 2 Execution model of XMP.

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 a **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 **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 such ways as global communication constructs and **coarray** assignments.

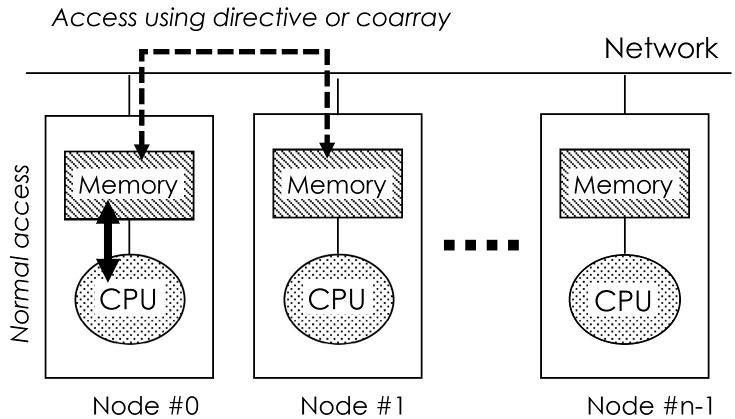

Fig. 3 Data model of XMP.

1.4 Programming Models

1.4.1 Partitioned Global Address Space

XMP can be classified as a *partitioned global address space (PGAS)* language, such as Co-Array Fortran [1], Unified Parallel C [2], and Chapel [3].

In such PGAS languages, 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, global-view and local-view, 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 serial version of a program, the programmer parallelizes it in a data-parallel style by adding directives with minimum modification. Based on this model, the programmer specifies 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 of distributed data, and move sections of distributed data globally. Note that the programmer must specify explicitly communication to make all data references in their program local using appropriate directives.

In many cases, the XcalableMP program following to the global-view programming model is based on a serial program, and it can produce the same result, regardless of the number of **nodes** (Fig. 4).

There are three groups of directives for this model:

- *Data mapping,* which specifies the data distribution and mapping to **nodes**
- *Work mapping (parallelization),* which specifies the work distribution and mapping to **nodes**.
- *Communication and synchronization,* which specify how a **node** communicates and synchronizes with the other **nodes**.

Because these directives are ignored as a comment by the compilers of base languages (Fortran and C), an XcalableMP program can usually be compiled by them to ensure that they run properly.

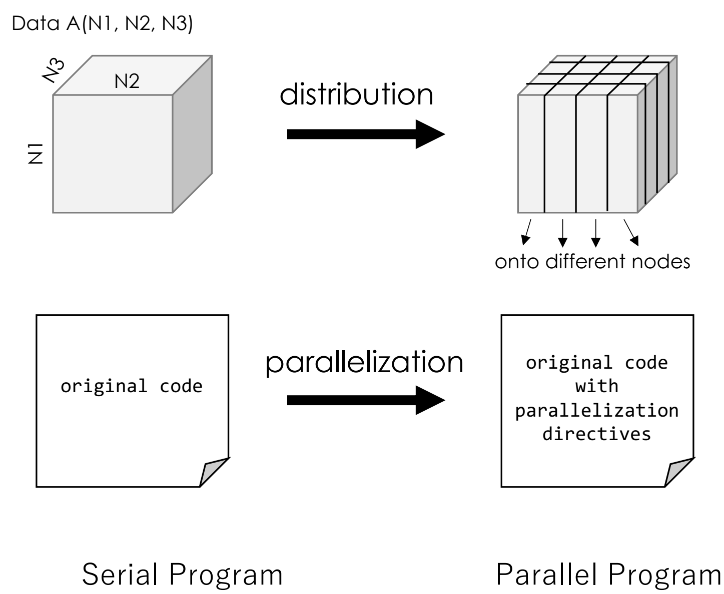

Fig. 4 Parallelization based on the global-view programming model.

1.4.3 Local-view Programming Model

The local-view programming model is suitable for programs that implement an algorithm and a remote data reference that are to be executed by each **node** (Fig. 5).

For this 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 explicitly specify data on which **node** 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 read 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 write the value to the remote memory (i.e. the *put* operation) is issued by the **executing node**.

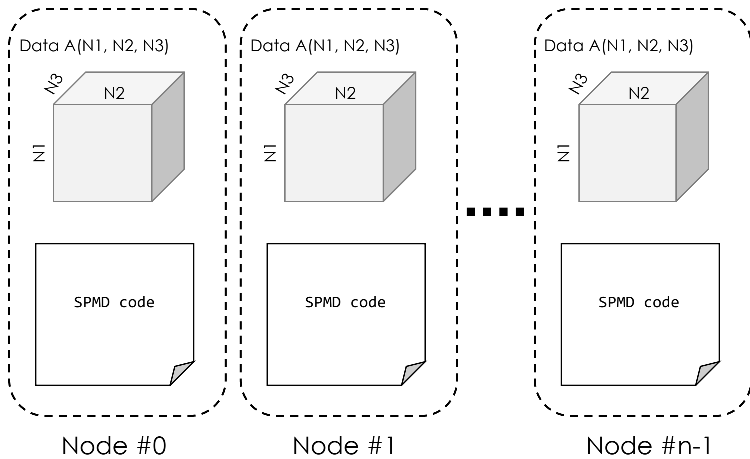

Fig. 5 Local-view programming model.

1.4.4 Mixture of Global View and Local View

In the global-view model, **nodes** are used to distribute data and works. In the local-view model, **nodes** are used to address remote data in the **coarray** notation. In application programs, the programmers should choose an appropriate data model according to the characteristics of their program. Fig. 6 illustrates the global view and the local view of data.

Data can have both a global view and a local view, and can be accessed in both of the views. XcalableMP provides a directive to give the **local** name (alias) to **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 a program by using explicit remote data access in the local-view programming model.

```
!$xmp nodes P(4)
!$xmp template T(100)
!$xmp distribute T(block) onto P

      real G(80, 100)        ! global variable
!$xmp align G(*,i) with T(i)

      real L(50, 40)         ! local variable (default)
```

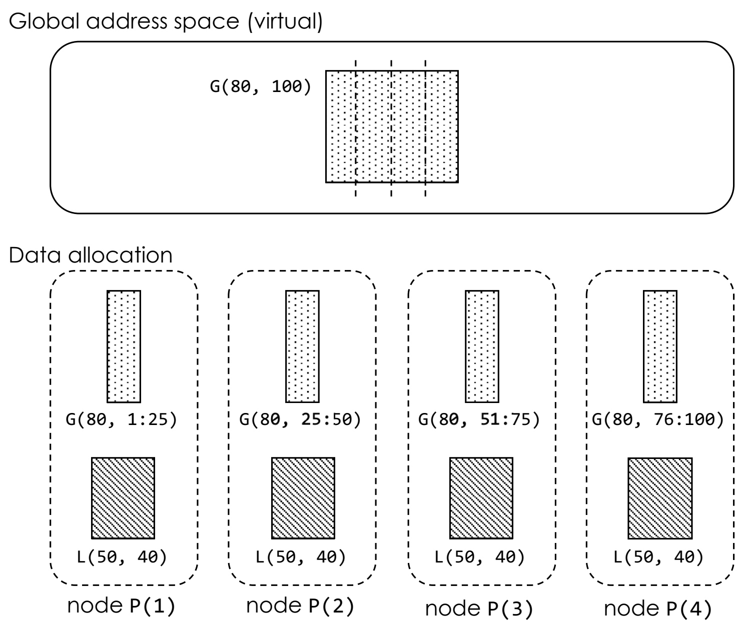

Fig. 6 Global view and local view.

1.5 Base Languages

The XcalableMP language specification is defined on the basis of 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, that is, an *array section* or *subarray*, can be put in an *array assignment statement*, which is described in 1.5.2, and some XcalableMP constructs. An array section is built from a subset of the elements of an array, which is specified by a sequence of square-bracketed integer expressions or *triplets*, which are in the form of:

[*base*] : [*length*] [: *step*]

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. The same applies to negative *step* too.

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

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[:]`	array section of the whole of `A`

1.5.2 Array Assignment Statement in XcalableMP C

In XcalableMP C, the base language C is also extended so that it supports array assignment statements just as Fortran does.

With such statement, 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 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 other array section(s) in the expression or on the left-hand side, 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.

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

XcalableMP C

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

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 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 a *node array*, which is an array-like arrangement of **nodes** in a **node set**. A **node array** can be multi-dimensional.

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

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

The `nodes` directive declares a one-dimensional **node array** `p` that includes four nodes. In XMP/C, it is zero-based and consists of `p[0]`, `p[1]`, `p[2]`, and `p[3]`. In XMP/Fortran, it is one-based and consists of `p(1)`, `p(2)`, `p(3)`, and `p(4)`.

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

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

The `nodes` directive declares two-dimensional **node array** `p` that includes six nodes. In XMP/C, it consists of `p[0][0]`, `p[0][1]`, `p[0][2]`, `p[1][0]`, `p[1][1]`, and `p[1][2]`. In XMP/Fortran, it 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 array** follows that of a normal array in the base language, C or Fortran.

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

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

An asterisk can be specified as the size in the `nodes` directive to declare a *dynamic* **node array**. In the above code, one-dimensional dynamic **node array** `p` is declared with an asterisk as the size. The actual size of a dynamic **node array** is determined at runtime to fit the size of the current executing node set. For example, when the programmer runs the sample code with three nodes, the **node array** `p` includes three nodes.

They can also declare multi-dimensional dynamic **node arrays** with an asterisk.

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

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

When the programmer runs the sample code with 12 nodes, the **node array** `p` has a shape of 4x3, in C, or 3x4, in Fortran.

Note: The user can put an asterisk only in the last dimension, in XMP/Fortran, or the first dimension, in XMP/C of the **node array**.

Hint: The dynamic **node array** may interfere with compiler optimizations. In general, programs with static ones achieve better performance.

The programmer can declare a node subarray derived from an existing node array. Node subarrays can be used, for example, 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 array** p including 16 nodes is declared. In line 2, a node subarray q corresponding to the first half of p is declared. In line 3, a two-dimensional node subarray r corresponding to the latter half of p is declared.

The programmer can declare a n-dimensional node subarray derived from a m-dimensional one.

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 array** p including 4x2 nodes is declared. In line 2, a node subarray row derived from a single row of p is declared. In line 3, a node subarray col derived from a single column of p is declared.

A colon represents a triplet which indicate all possible indices in the dimension. An asterisk indicate the index of the current executing node in the dimension. For example, col[2] corresponds to p[0][0:2] on nodes p[0][0] and p[0][1], and to p[1][0:2] on nodes p[1][0] and p[1][1] in XMP/C. Similarly, col(2) corresponds to p(1:2,1) on nodes p(1,1) and p(2,1), and to p(1:2,2) on nodes p(1,2) p(2,2) in XMP/Fortran.

In XMP/C, row[0] corresponds to p[0][0] and p[0][1] on p[:][0] and p[:][1], respectively; col[0] corresponds to p[0][0], p[1][0], p[2][0], and p[3][0] on p[0][:], p[1][:], p[2][:], p[3][:], respectively. In XMP/Fortran, row(1) corresponds to p(1,1) and p(2,1) on p(1,:) and p(2,:), respectively; col(1) corresponds to p(1,1), p(1,2), p(1,3), and p(1,4) on p(:,1), p(:,2), p(:,3), p(:,4), respectively.

Note: The semantics of an asterisk in a node reference is different from that in a declaraion.

2.2 template Directive

The template directive declares a *template*, which is a virtual array that is used as a “template” of parallelization in the programs and to be distributed onto a **node array**.

XcalableMP C
```
#pragma xmp template t[10]
```

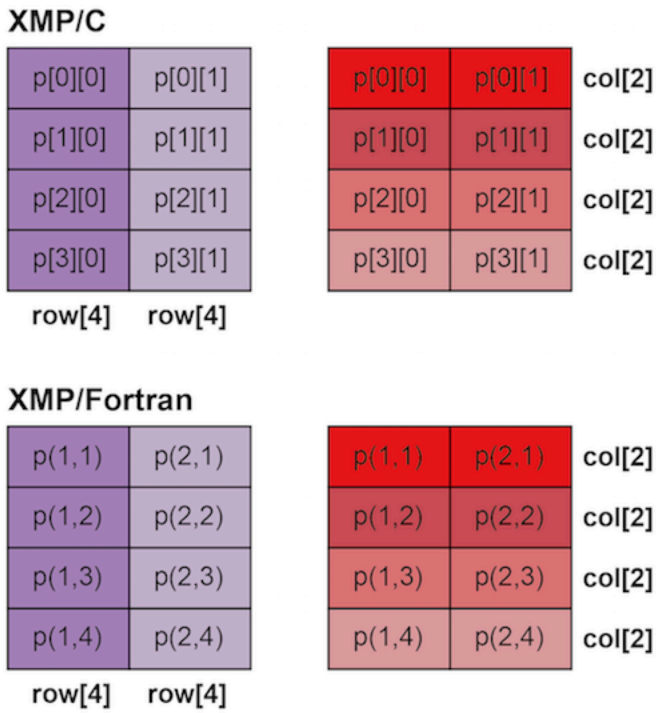

Fig. 7 Node subarrays.

XcalableMP Fortran
```
!$xmp template t(10)
```

This `template` directive declares a one-dimensional **template** `t` having ten elements. **Templates** are indexed in the similar manner to arrays in the base languages. For the above examples, the **template** `t` is indexed from zero to nine (i.e. `t[0]` $\cdots$ `t[9]`), in XMP/C, or one to ten (i.e. `t(1)` $\cdots$ `t(10)`), in XMP/Fortran.

Hint: In many cases, a template should be declared to have the same shape as your target array.

XcalableMP C
```
#pragma xmp template t[10][20]
```

XcalableMP Fortran
```
!$xmp template t(20,10)
```

The `template` directive declares a two-dimensional **template** `t` that has 10x20 elements. In XMP/C, `t` is indexed from t[0][0] to t[9][19], and, in XMP/Fortran, from `t(1,1)` to `t(20,10)`.

XcalableMP C
```
#pragma xmp template t[:]
```

XcalableMP Fortran
```
!$xmp template t(:)
```

In the above examples, a colon instead of an integer is specified as the size to declare a one-dimensional dynamic **template** `t`. The colon indicates that the size of the **template** is not fixed and to be fixed at runtime by the `template_fix` construct (Sec. 2.6).

2.3 `distribute` Directive

The `distribute` directive specifies a distribution of the target **template**. Either of *block*, *cyclic*, *block-cyclic*, or *gblock* (i.e. uneven block) can be specified to distribute a dimension of a **template**.

2.3.1 Block Distribution

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

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

The target template `t` is divided into contiguous blocks and distributed among nodes in the **node array** `p`. Let's suppose that the size of the **template** is N and the number of nodes is K. If N is divisible by K, a block of size N/K are assigned to each node; otherwise, a block of size $ceil(N/K)$ is assigned to each of $N/ceil(N/K)$ nodes, a block of size $mod(N,K)$ to one node, and no block to $(K-N/ceil(N/K)-1)$ nodes. The block distribution is useful for regular computations such as a stencil one.

Note: The function $ceil(x)$ returns a minimum integer value greater than x, and $mod(x,y)$ returns x modulo y.

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
```

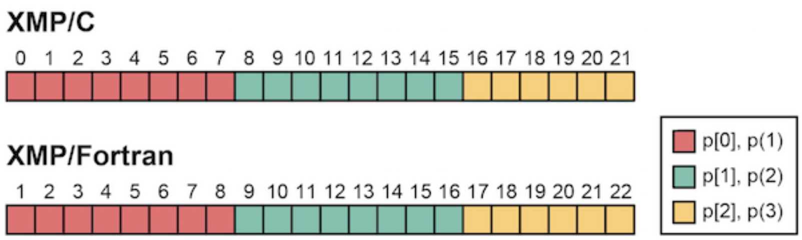

Fig. 8 Block distribution.

Since $ceil(22/3)$ is 8, eight elements are allocated on each of `p[0]` and p[1], and the remaining six elements are allocated on `p[2]`.

2.3.2 Cyclic Distribution

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

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

The target **template** `t` is divided into chunks of size one and distributed among nodes in the **node array** `p` in a round-robin manner. The cyclic distribution is usefull for the case where the load on each element of the **template** is not balanced.

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
```

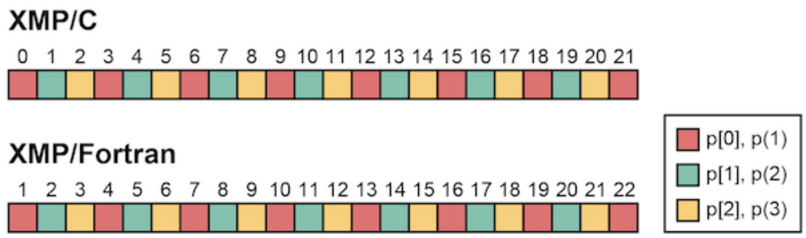

Fig. 9 Cyclic distribution.

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

The target **template** `t` is divided into chunks of size `w` and distributed among nodes in the **node array** `p` in a round-robin manner. The block-cyclic distribution is usefull for the case where the load on each element of the **template** is not balanced but the locality of the elements is required.

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
```

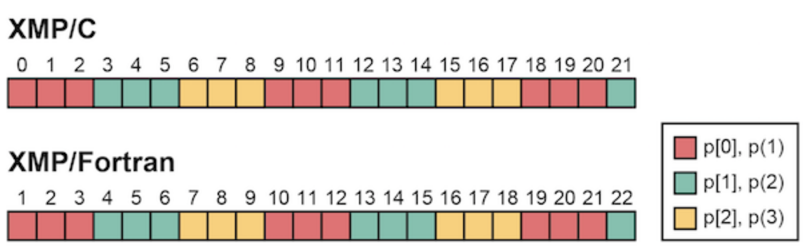

Fig. 10 Block-cyclic distribution.

2.3.4 Gblock Distribution

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

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

The target **template** t is divided into contiguous blocks of size W[0], W[1], $\cdots$, in XMP/C, or W(1), W(2), $\cdots$, in XMP/Fortran, and distributed among nodes in the **node array** p. The array W is called a mapping array. The programmer can specify irregular (uneven) block distribution with the gblock format.

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
```

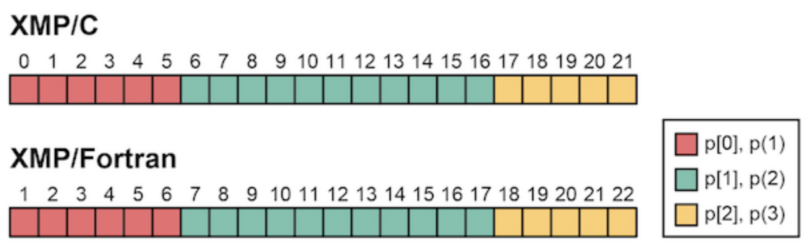

Fig. 11 Gblock distribution.

The programmer can specify an asterisk instead of a mapping array in the gblock distribution to defer fixing the actual distribution. In such a case, the actual distribution will be fixed at runtime by using `template_fix` construct.

2.3.5 Distribution of Multi-dimensional Templates

The programmer can distribute a multi-dimensional **template** onto a **node array**.

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 the distribution of a two-dimensional **template** `t` onto a two-dimensional **node array** `p`. Each dimension of the **template** is divided in a block manner and each of the rectangular regtion is assigned to a node.

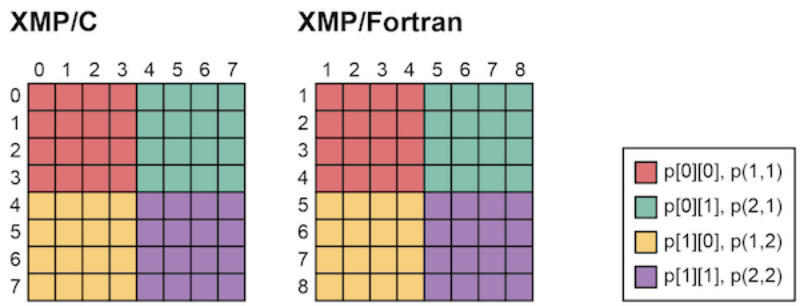

Fig. 12 Example of multi-dimensional distribution (1).

The programmer can specify a different distribution format in each of the dimension of a **template**.

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 is specified in a `distribute` directive as a distribution format, the target dimension is “non-distributed.” In the following example, the first dimension is distributed in a block manner and the second dimension is non-distributed.

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
```

Fig. 13 Example of multi-dimensional distribution (2).

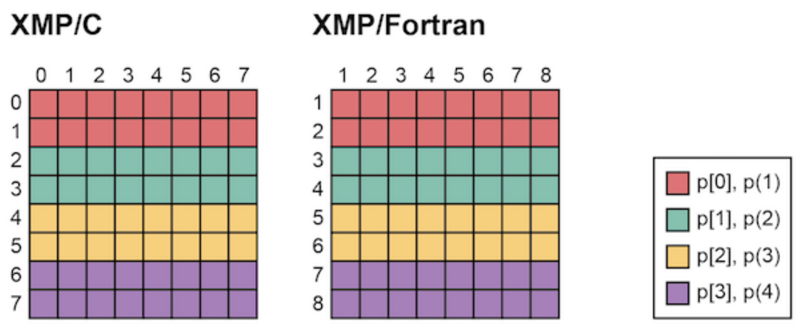

Fig. 14 Example of multi-dimensional distribution (3).

2.4 `align` Directive

The `align` directive specifies that an array is to be mapped in the same way as a specified **template**. In other words, an `align` directive defines the correspondece of elements between an array and a **template**, and each of the array element is allocated on the node where the corresponding **template** element is allocated.

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 array a is decomposed and laid out so that each element a(i) is colocated with the corresponding **template** element t(i).

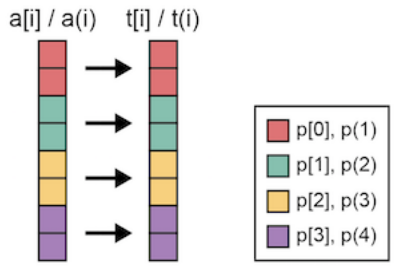

Fig. 15 Example of array alignment (1).

The align directive can also 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)
```

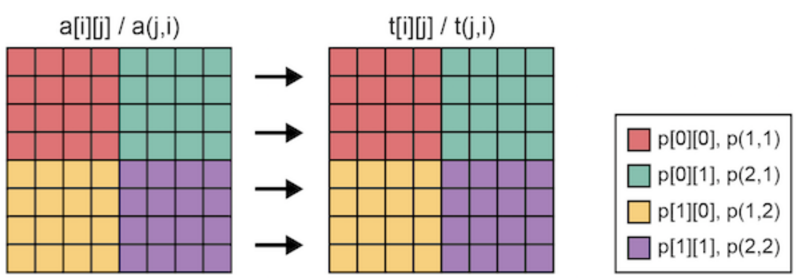

Fig. 16 Example of array alignment (2).

The programmer can align an n-dimensional array with an m-dimensional **template** for $n > m$.

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 is specified as a subscript in a dimension of the target array in the `align` directive, the dimension is "collapsed" (i.e. not distributed). In the sample program above, the first dimension of the array a is distributed onto the **node array** p while the second dimension is collapsed.

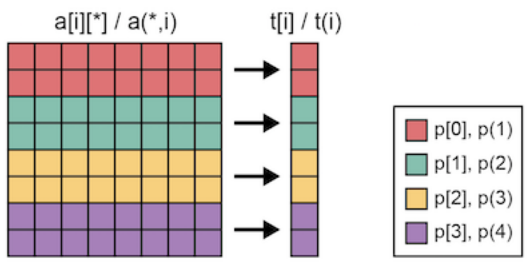

Fig. 17 Example of array alignment (3).

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

The programmer also can align an n-dimensional array with an m-dimensional template for $n < m$.

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 is specified as a subscript in a dimension of the target **template** in the `align` directive, the array will be “replicated” along the axis of the dimension.

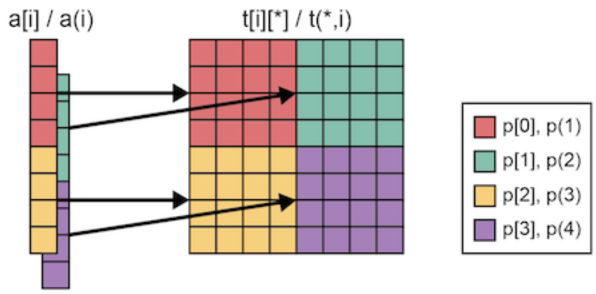

Fig. 18 Example of array alignment (4).

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

2.5 Dynamic Allocation of Distributed Array

This section explains how **distributed** (i.e. **global**) **arrays** are allocated at runtime. The basic procedure is common in XMP/C and XMP/Fortran with a few specific difference.

XcalableMP C
```
#pragma xmp nodes p[4]
#pragma xmp template t[N]
#pragma xmp distribute t[block] onto p
float *a;
#pragma xmp align a[i] with t[i]
  :
a = xmp_malloc(xmp_desc_of(a), N);
```

In XMP/C, first, declare a pointer of the type of the target array; second, align it as if it were an array; finally, allocate memory for it with the `xmp_malloc()` function. `xmp_desc_of()` is an intrinsic/builtin function that returns the descriptor of the XMP object (i.e. **nodes**, **templates**, or **global arrays**) specified by the argument.

XcalableMP Fortran
```
!$xmp nodes p(4)
!$xmp template t(N)
```

```
!$xmp distribute t(block) onto p
real, allocatable :: a(:)
!$xmp align a(i) with t(i)

allocate(a(N))
```

In XMP/Fortran, first, declare an allocatable array; second, align it; finally, allocate memory for it with the `allocate` statement.

For multi-dimensional arrays, the procedure is the same as that for one-dimensional ones, as follows:

XcalableMP C

```
#pragma xmp nodes p[2][2]
#pragma xmp template t[N1][N2]
#pragma xmp distribute t[block][block] onto p
float (*a)[N2];
#pragma xmp align a[i][j] with t[i][j]
   :
a = (float (*)[N2])xmp_malloc(xmp_desc_of(a), N1, N2);
```

XcalableMP Fortran

```
!$xmp nodes p(2,2)
!$xmp template t(N2,N1)
!$xmp distribute t(block,block) onto p
real, allocatable :: a(:,:)
!$xmp align a(j,i) with t(j,i)
   :
allocate(a(N2,N1))
```

Note: If the size of the **template** is not fixed until runtime, the programmer have to fixe it at runtime with the `template_fix` construct.

2.6 `template_fix` Construct

The `template_fix` construct fixes the shape and/or the distribution of an unfixed **template**.

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))
```

In the above sample code, first, a **template** `t` whose size is unfixed ("`:`") is declared; second, a pointer `a`, in XMP/C, or an allocatable array `a`, in XMP/Fortran, is aligned with the **template**; third, the size of the **template** is fixed with a `template_fix` construct; finally, the pointer or the allocatable array is allocated with the `xmp_malloc()` builtin function in XMP/C or the `allocate` statement in XMP/Fortran, respectively.

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

This construct can also be used to fix 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 and tasks Construct

The `task` construct defines a *task* that is executed by a specified **node set**. The `tasks` construct asserts that surrounding `task` constructs can be executed in parallel.

3.1.1 task Construct

When a **node** encounters a *task* construct at runtime, it executes the associated block (called a task) if it is included by the **node set** specified by the on clause; otherwise, it skips the execution of the block.

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]` invokes the `printf()` function, and `p[1]` outputs “1: Hello” in XMP/C; `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. Let’s consider inserting a `bcast` construct into the 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 `on` clause of the `task` construct. Thus, the **node** `p[1]` broadcasts the value of `num` 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** `q` 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 in 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());
```

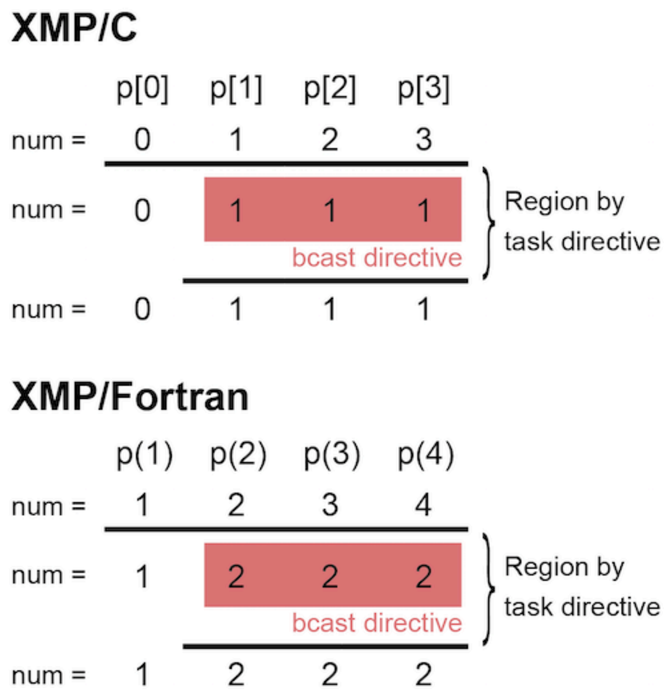

Fig. 19 Example of `task` construct (1).

```
}

  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 a `task` construct unless it is surrounded by a `tasks` construct. Therefore, 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

Let's 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**.

In such a case, the programmer must specify a `tasks` construct surrounding the tasks is needed to execute them 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();
```

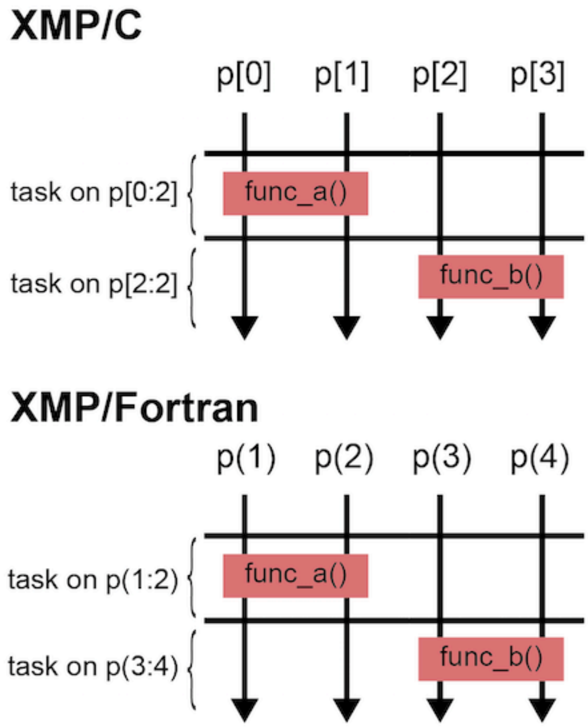

Fig. 20 Example of `task` construct (2).

```
}
}
```

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
```

Because the **node set**s specified by the `on` clauses of the `task` constructs surrounded by a `tasks` construct are disjoint, they can be executed in parallel.

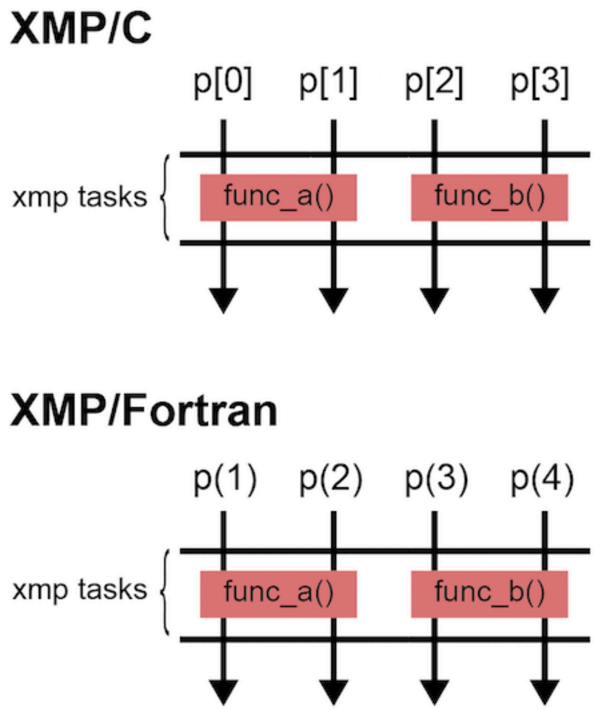

Fig. 21 Example of `tasks` construct.

3.2 `loop` Construct

The `loop` construct is used to parallelize a loop.

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

XcalableMP Fortran
```
!$xmp loop on t(i)
  do i = 1, 10
    a(i) = i
  end do
```

The `loop` directive above specifies that the iteration `i` of the following loop is executed by the **node** that owns the **template** element `t[i]` or `t(i)`, which is specified in the `on` clause.

Such a loop must satisfy the following two conditions:

1. 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.

2. Elements of **distributed arrays**, if any, are accessed only by the bf node(s) that own(s) the elements.

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
```

Then, is it possible to parallelize the loops in the below example where the loop bounds are shrunk from the above?

XcalableMP C

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

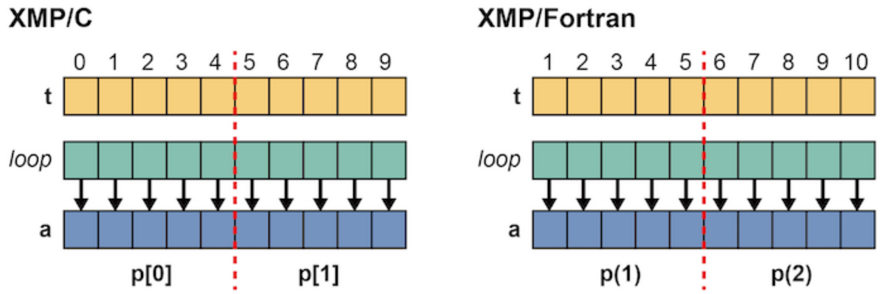

Fig. 22 Example of `loop` construct (1).

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.

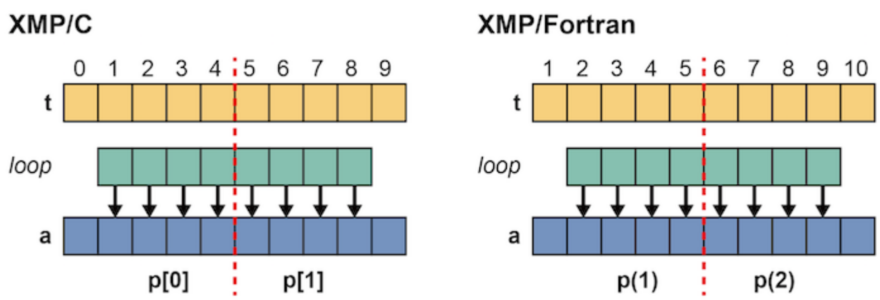

Fig. 23 Example of `loop` construct (2).

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.

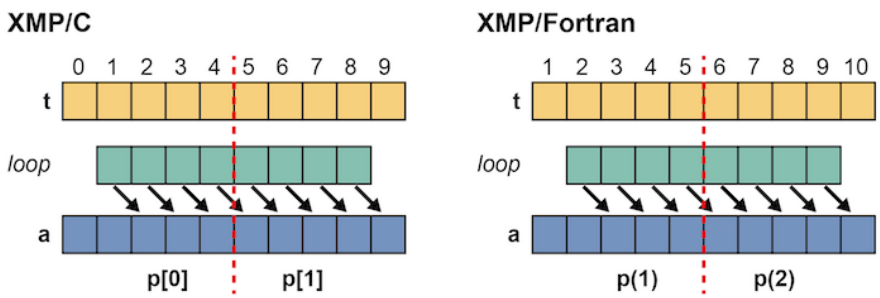

Fig. 24 Example of `loop` construct (3).

3.2.1 Reduction Computation

The serial programs below are examples of a 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

```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 just by adding a `loop` directive, the value of the variable `sum` varies from **node** to **node** because it is calculated separately on each **node**. The value should be *reduced* to produce the right result.

XcalableMP 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
```

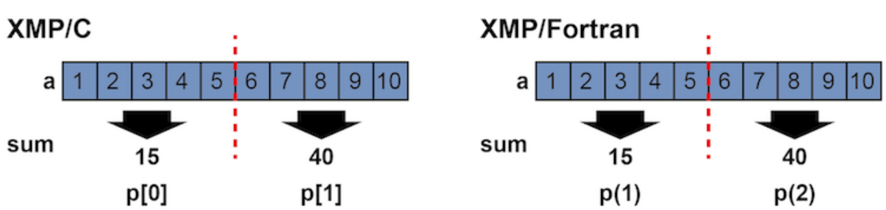

Fig. 25 Example of reduction computation (1).

Then, to correct the error in the above code, add a `reduction` clause to the `loop` directive as follows.

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 computation are specified in a `reduction` clause. In the above examples, a "+" operator and a target variable `sum` are specified for the reduction computation to produce a total sum among **nodes**.

Operations that can be specified as an operator in a `reduction` clause are limited to the following associative ones.

C

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

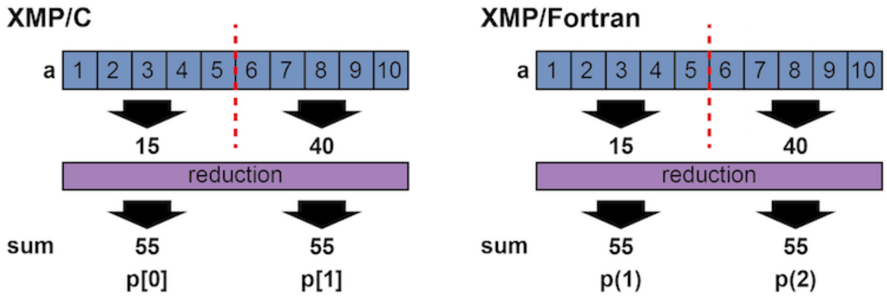

Fig. 26 Example of reduction computation (2).

```
min
firstmax
firstmin
lastmax
lastmin
```

Fortran

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

Note: The total result is calculated by combining the partial results on all **nodes**. The ordering of the combination is unspecified. Hence, if the target variable is a type of floating point (e.g. `float` in XMP/C or `real` in XMP/Fortran), the difference of the order can make a little bit difference in the result value from that in the original serial execution.

3.2.2 Parallelizing Nested Loop

Parallelization of nested loops can be specified similarly to a single one, as follows.

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.

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 as the arrays in the following statement. The right-hand side value in this construct must be identical among all **nodes** because the `array` construct is a **global** (i.e. collective) operation.

---

4 Data Communication

4.1 shadow Directive and reflect Construct

Stencil computation frequently appears in scientific simulation programs, where, to update an array element `a[i]`, its neighboring elements `a[i-1]` and `a[i+1]` are referenced. If `a[i]` is on the boundary region of a block-**distributed array** on a **node**, `a[i+1]` may reside on another (neighboring) **node**.

Since it involves large overhead to copy `a[i+1]` from the neighboring **node** to update each `a[i]`, a technique of copying collectively the elements on the neighboring **node** to the area added to the **distributed array** on each **node** is usually adopted. In XMP, such additional area is called "shadow."

4.1.1 Declaring Shadow

Shadow areas can be declared with the `shadow` directive. In the example below, an array `a` has shadow areas of width 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)
```

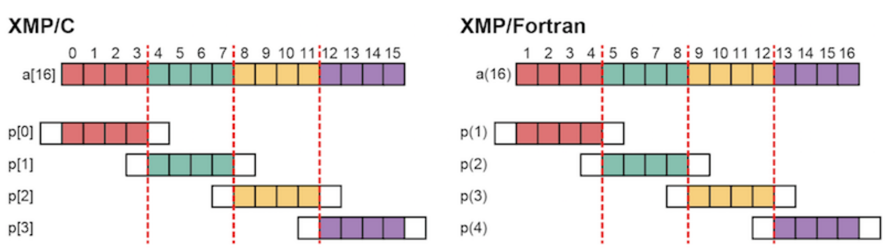

Fig. 27 Example of `shadow` directive (1).

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

Note: Arrays distributed in a cyclic manner cannot have shadow.

In 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)
```

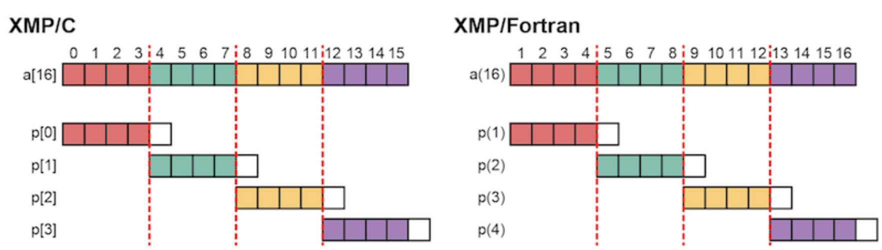

Fig. 28 Example of shadow directive (2).

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 Updating Shadow

To copy data to shadow areas from neighboring **nodes**, use the `reflect` construct. In the example below, the shadow areas of an array `a` that are of width one on both the upper and lower bounds are updated.

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
```

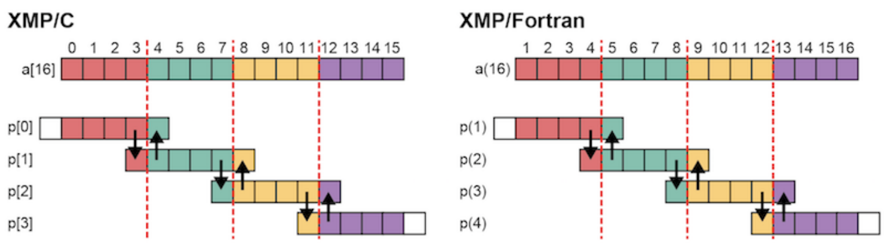

Fig. 29 Example of `reflect` construct (1).

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)`.

The default behavior of a `reflect` directive is to update the whole of the shadow area declared by the `shadow` directive. However, there are some cases where a specific part of the shadow area is to be updated to reduce the communication cost at 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)
```

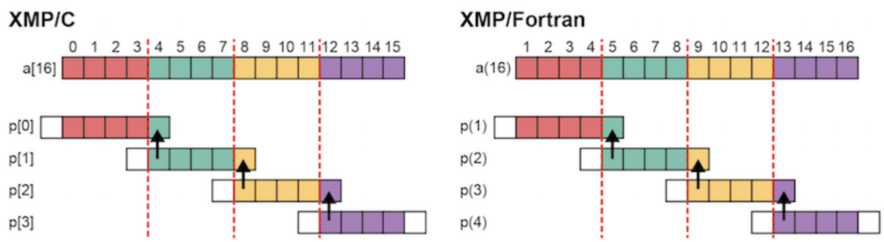

Fig. 30 Example of `reflect` construct (2).

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** because `reflect` is a collective operation.

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 periodic boundary conditions are used in the computation.

To update such areas, add a `periodic` qualifier into the `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)
```

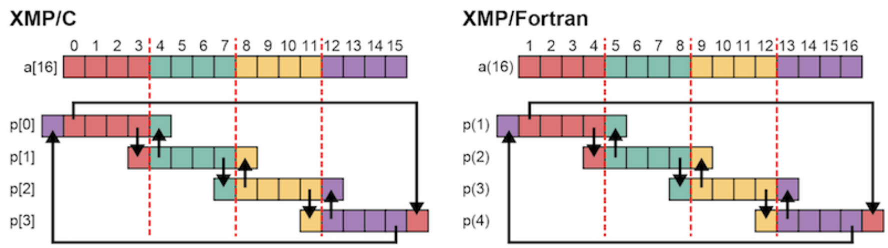

Fig. 31 Example of periodic `reflect` construct.

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)`.

The `shadow` directive and `reflect` construct can be applied to arrays distributed in 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 data from the surrounding eight **nodes** to update its shadow areas. 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 the `orthogonal` clause to a `reflect` construct.

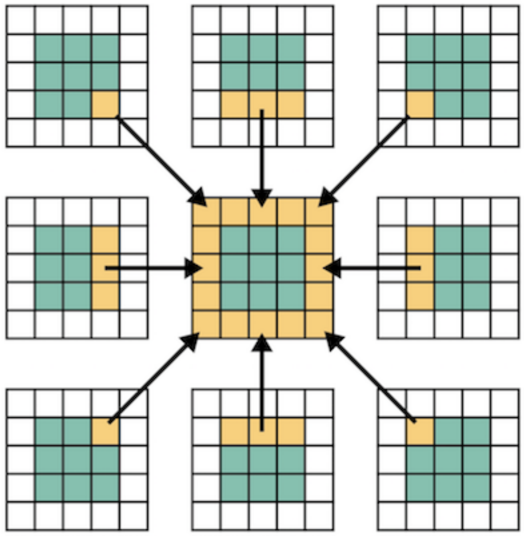

Fig. 32 Example of multi-dimensional shadow (1).

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
```

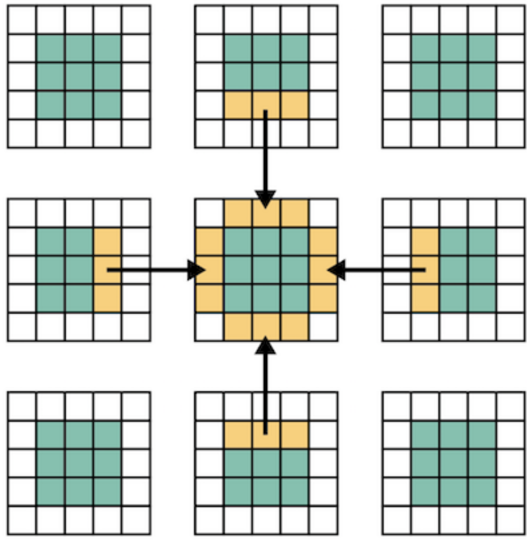

Fig. 33 Example of multi-dimensional shadow (2).

```
real :: a(9,9)
!$xmp align a(*,i) with t(i)
!$xmp shadow a(0,1)
  :
!$xmp reflect (a)
```

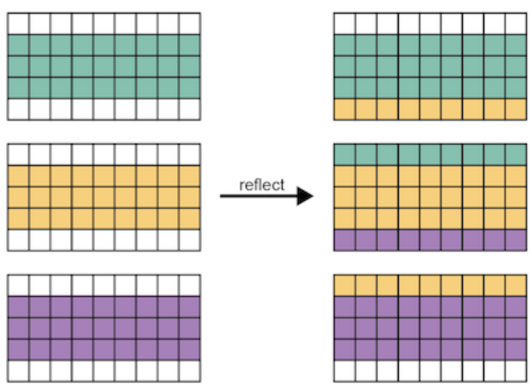

Fig. 34 Example of multi-dimensional shadow (3).

For the array `a`, 0 is specified as the shadow width in non-distributed dimensions.

4.2 `gmove` Construct

The programmaers can specify a communication of **distributed arrays** in the form of assignment statements by using the `gmove` construct. In other words, with the `gmove` construct, any array assignment between two arrays (i.e. *global data movement*) that may involve inter-node communication can be specified.

There are three modes of `gmove`; "collective mode," "in mode," and "out mode."

4.2.1 Collective Mode

The global data movement involved by a *collective* `gmove` is performed collectively, and results in implicit synchronization among the **executing nodes**.

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)
```

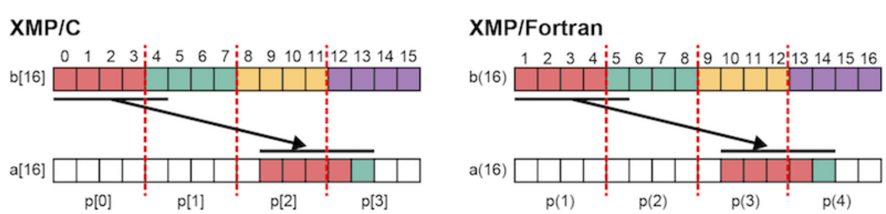

Fig. 35 Collective `gmove` (1).

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)`.

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)
```

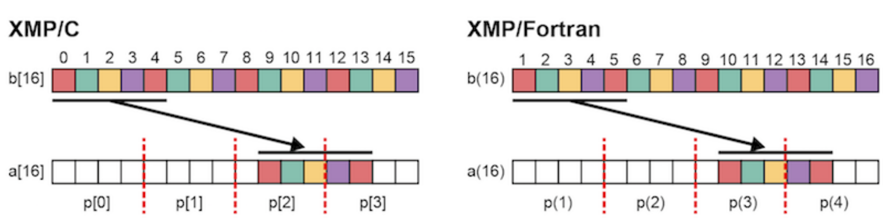

Fig. 36 Collective `gmove` (2).

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.

By using this method, the **distribution** of an array can be “changed” during computation.

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(:)
```

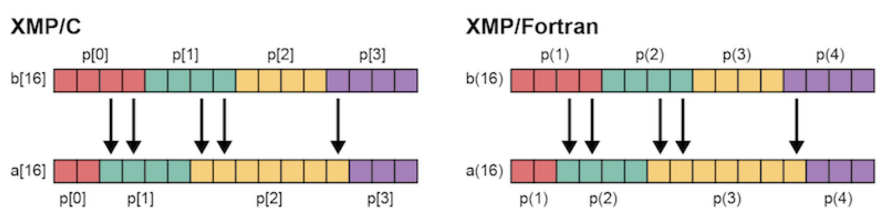

Fig. 37 Collective gmove (3).

In this example, the elements of an array b that is distributed in a block manner are copied to the corresponding elements of an array a that is distributed in a generalized-block manner. For the arrays a and b, communication occurs if corresponding elements reside in different **nodes** (arrows illustrate communication between **nodes** in the figures).

In the assignment statement, if a scalar (i.e. one element of an array or a variable) is specified on the right-hand side and an array section are specified on the left-hand side, a broadcast communication occurs for it.

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)
```

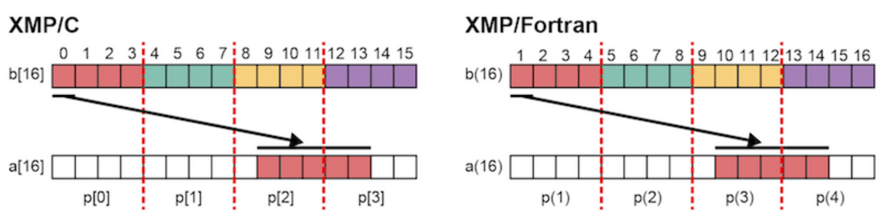

Fig. 38 Collective gmove (4).

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

Not only **distributed arrays** but also replicated arrays can be specified 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];
```

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)
```

In this example, a replicated array b is locally copied to **distributed array** a without communication.

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] on p[0], b[0][2:2] of p[1], b[0][4:2] on p[2] and b[0][6:2] on p[3] are copied to a[0][:] on p[0]. Similarly, in XMP/Fortran, b(1:2,1) on p(1), b(3:4,1) of p(2), b(5:6,1) on p(3) and b(7:8,1) on p(4) are copied to a(:,1) on p(1).

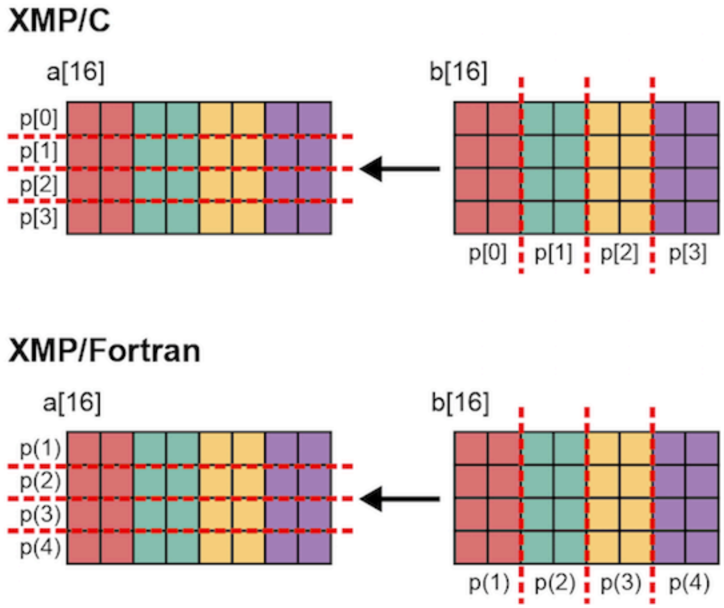

Fig. 39 Collective gmove (4).

4.2.2 In Mode

The right-hand side data of the assignment, all or part of which may reside outside the **executing node set**, can be transferred from its owner **nodes** to the **executing nodes** with an *in* gmove.

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 four **nodes** into two sets, the first-half and the second-half. A `gmove` construct that is in an *in* mode copies data using a *get* operation from the second-half **node** to the first-half **node**.

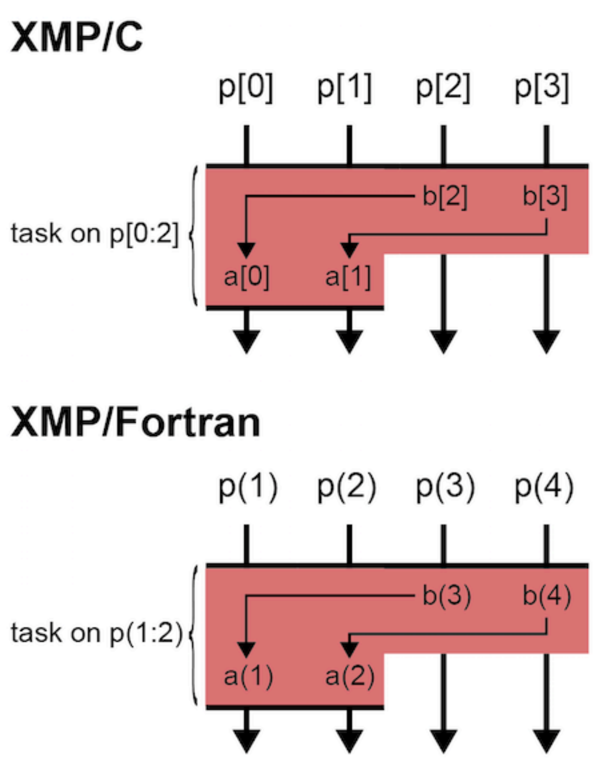

Fig. 40 In `gmove`.

4.2.3 Out Mode

For the left-hand side data of the assignment, all or part of which may reside outside the **executing node set**, the corresponding elements can be transferred from the **executing nodes** to its owner **nodes** with an *out* `gmove` construct.

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
```

A `gmove` construct that is in *out* mode copies data using a *put* communication from the first-half **nodes** to the second-half **nodes**.

4.3 barrier Construct

The `barrier` construct executes a barrier synchronization.

XcalableMP C
```
#pragma xmp barrier
```

XcalableMP Fortran
```
!$xmp barrier
```

You can specify a **node set** on which the barrier synchroniation is to be performed by using the `on` clause. In the below example, a barrier synchronization is performed among the first two **nodes** of `p`.

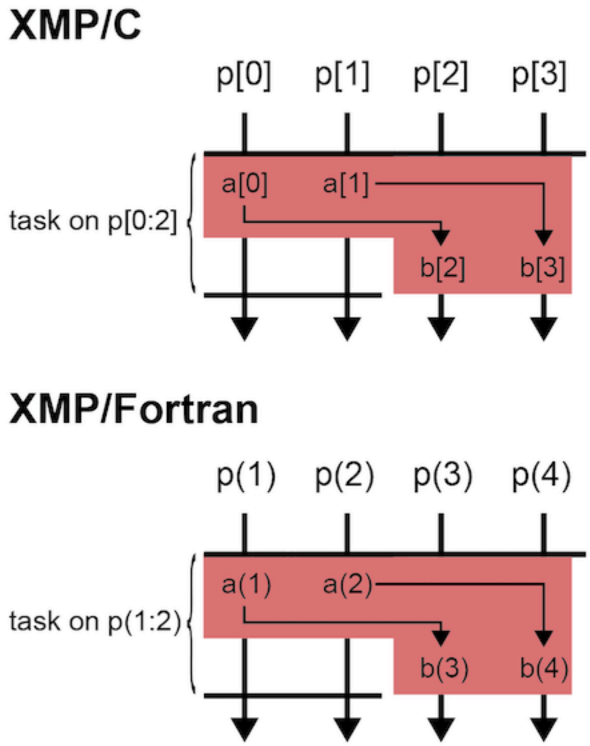

Fig. 41 Out gmove.

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

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

4.4 reduction Construct

This construct performs a *reduction* operation. It has the same meaning as the `reduction` clause of the `loop` construct, but this construct can be specified anywhere as an *executable* construct.

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)
```

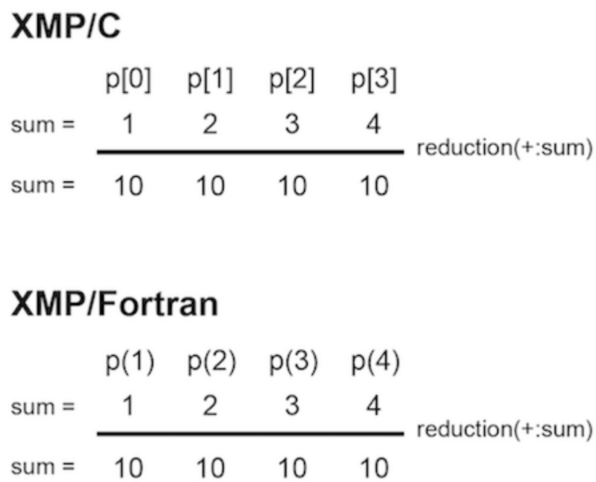

Fig. 42 `reduction` construct (1).

You can specify the **executing node set** by using the `on` clause. In the below example, only the values on the last two of the four **nodes** are targeted by the `reduction` construct.

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 operators you can use in the `reduction` construct are as follows:

XcalableMP C

```
+
*
-
```

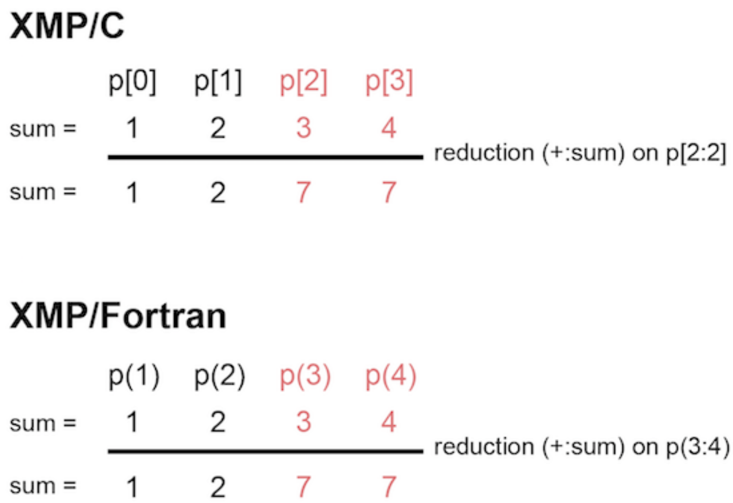

Fig. 43 `reduction` construct (2).

```
&
|
^
&&
||
max
min
```

XcalableMP Fortran

```
+
*
-
.and.
.or.
.eqv.
.neqv.
max
min
iand
ior
ieor
```

Note: In contrast to the `reduction` clause of the `loop` construct, which precedes loops, the `reduction` construct does not accept operators of `firstmax`, `firstmin`, `lastmax`, and `lastmin`.

Note: Similar to the `reduction` clause, the `reduction` construct may generate slightly different results in a parallel execution from those in a sequential execution, because the results depends on the order of combining the value.

4.5 bcast Construct

The `bcast` construct broadcasts the values of the variables on the **node** specified by the `from` clause, that is, the *root node*, to the **node set** specified by the `on` clause. If there is no `from` clause, the first **node** of the **executing node set** is selected as the root **node**. If there is no `on` clause, the current **executing node set** of the construct is selected as the **executing node set**.

In the below example, the first **node** of the **node set** `p`, that is, `p[0]` or `p(1)`, is the root **node**.

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, the last **node**, that is, `p[3]` or `p(4)`, is 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 last three of four **nodes** are included by the **executing node set** of the `bcast` construct.

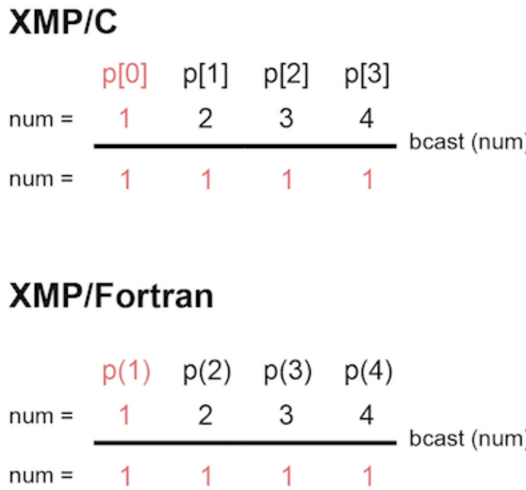

Fig. 44 `bcast` construct (1).

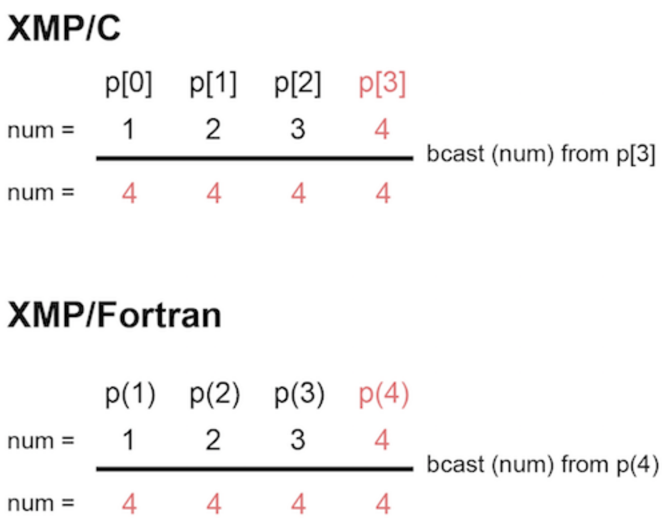

Fig. 45 `bcast` construct (2).

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)
```

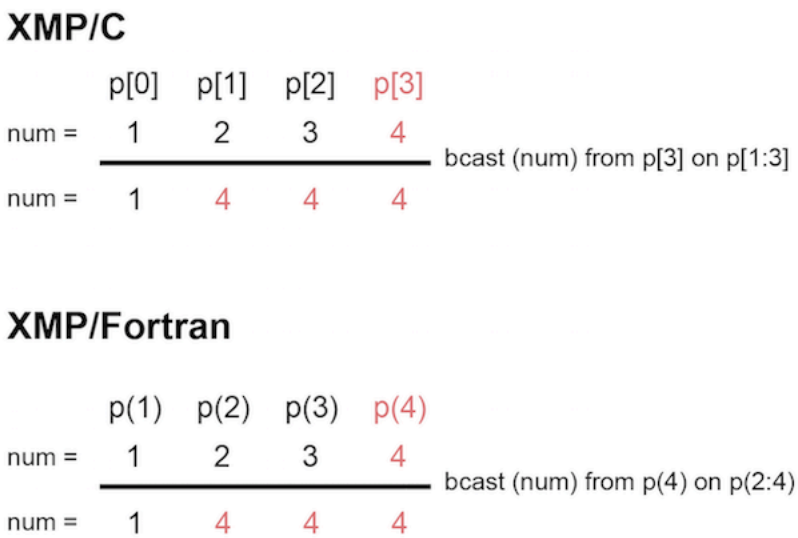

Fig. 46 `bcast` construct (3).

4.6 wait_async Construct

Communication directives (i.e. `reflect`, `gmove`, `reduction`, `bcast`, and `reduce_shadow`) can perform asynchronous communication if the `async` clause is added. The `wait_async` construct is used to guarantee the completion of such an 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. The completion of that communication is guaranteed with the `wait_async` construct having the same value as that of the `async` clause. Therefore, between the `bcast` construct and the `wait_async` constructs, you may not reference the target variable of the `bcast` directive.

Hint: Asynchronous communication can be overlapped with the following computation to hide its overhead.

Note: Expressions that can be specified as *tags* in the `async` clause are of type int, in XMP/C, or integer, in XMP/Fortran.

4.7 reduce_shadow Construct

The `reduce_shadow` directive adds the value of a shadow object to the corresponding data object of the array.

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)
```

For the above example, in XMP/C, `a[3]` on `p[0]` has a value of eight, and `a[4]` on `p[1]` has a value of ten. Similarly, in XMP/Fortran, `a(4)` of `p(1)` has a value of eight, and `a(5)` on `p(2)` has a value of ten.

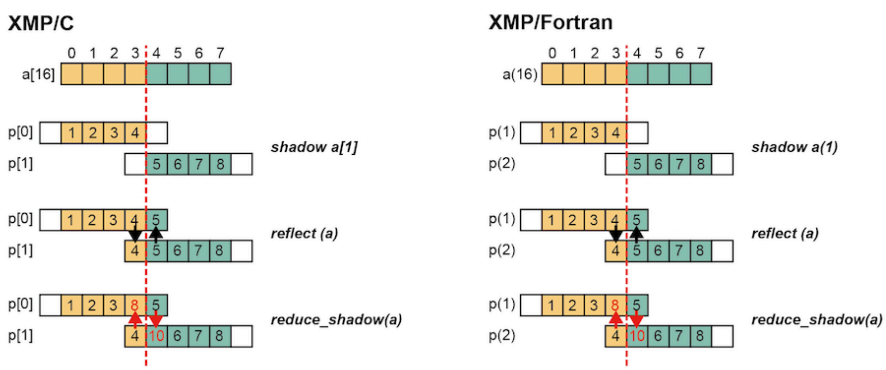

Fig. 47 `reduce_shadow` construct (1).

The programmers can add the `periodic` modifier to the `width` clause to reduce shadow objects to the corresponding data object periodically.

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]` on `p[0]` has a value of two, and `a[7]` on `p[1]` has a value of 16. Similarly, in XMP/Fortran, `a(1)` in `p(1)` has a value of two, and `a(8)` in `p(2)` has a value of 16.

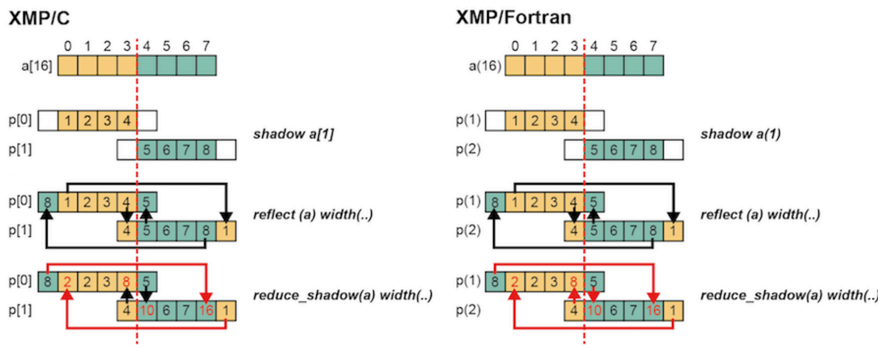

Fig. 48 `reduce_shadow` construct (2).

5 Local-view Programming

5.1 Introduction

The programmer can use **coarrays** to specify one-sided communication in the local-view model.

Depending on the environment, such one-sided communication might achieve better performance than global communication in the global-view model. However, it is more difficult and complicated to write parallel programs in the local-view model because the programmer must specify every detail of parallelization, such as data mapping, work mapping, and communication.

The **coarray** feature in XMP/Fortran is upward-compatibile with that in Fortran 2008; that in XMP/C is defined as an extension to the base language.

An execution entity in local-view XMP programs is referred to as an “image” while a **node** in global-view ones. These two words have almost the same meaning in XMP.

5.2 Coarray Declaration

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

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

In XMP/C, the programmer declares a **coarray** by adding “`:[*]`” after the array declaration. In XMP/Fortran, the programmer declares a **coarray** by adding “`[*]`” after the array declaration.

Note: Based on Fortran 2008, **coarrays** should have the same size among all images.

Coarrays can be accessed in expressions by remote images as well the local image.

5.3 Put Communication

When a **coarray** appears in the left-hand side of an assignment statement, it involves *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 in the square bracket specifies the target image index. The image index is zero-based, in XMP/C, or one-based, in XMP/Fortran. `xmpc_this_image()` in XMP/C and `this_image()` in XMP/Fortran return the current image index.

In the above example, in XMP/C, an image zero puts `b[3:3]` to `a[0:3]` on image one; in XMP/Fortran, an image one puts `b(3:5)` to `a(1:3)` on image two. The following figure illustrates the put communication performed in the example.

5.4 Get Communication

When a **coarray** appears in the right-hand side of an assignment statement, it involves *get* communication.

XcalableMP C

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

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

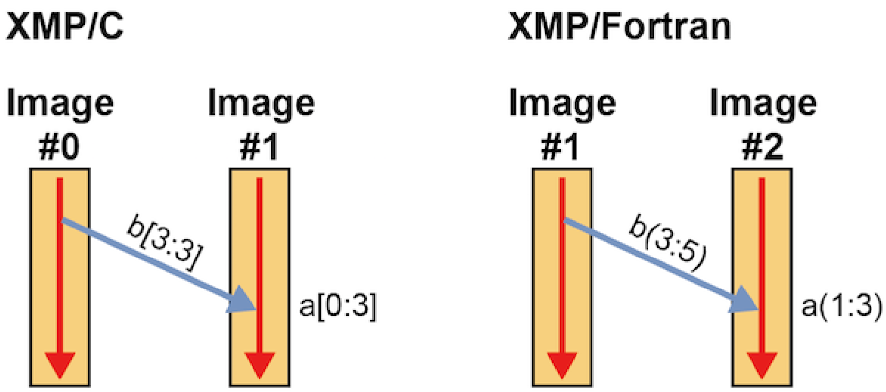

Fig. 49 Remote write to a coarray

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 example, in XMP/C, an image 0 gets `a[0:3]` from an image 1 and copies it to `b[3:3]`; in XMP/Fortran, an image 1 gets `a(1:3)` from an image 2 and copies it to `b(3:5)` of an image 1. The following figure illustrates the get communication performed in the example.

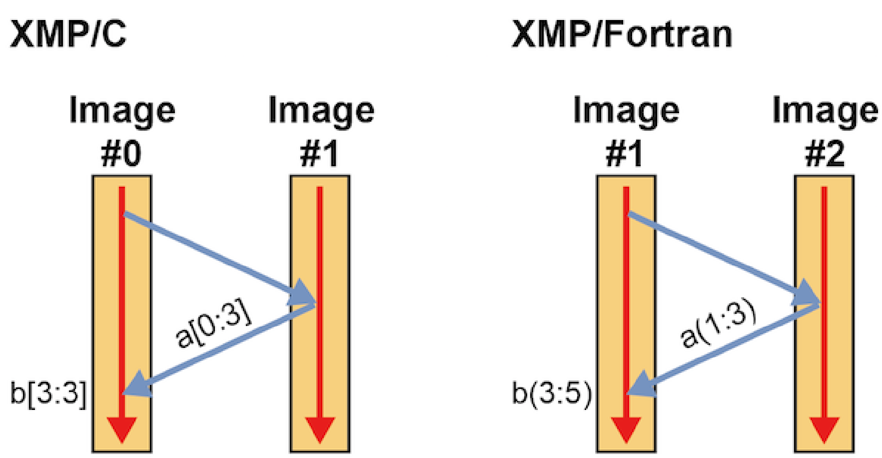

Fig. 50 Remote read from a coarray

Hint: As illustrated above, get communication involves an extra step to send a request to the target **node**. Put communication achieves better performance than get because there is no such extra step.

5.5 Synchronization

5.5.1 sync all

XcalableMP C

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

XcalableMP Fortran

```
sync all
```

At “sync all,” each image waits until all issued one-sided communication is complete and then performs barrier synchronization among the all images.

In the above example, the left image puts data to the right image and both **nodes** invoke `sync all`. When both **nodes** returns from it, the execution continues to the following satements.

5.5.2 sync images

XcalableMP C

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

XcalableMP Fortran

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

Each image in the specified image set waits until all one-sided communication issued is complete, and performs barrier synchronization among the images.

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)
```

5.5.3 sync memory

XcalableMP C

```
void xmp_sync_memory(int *status)
```

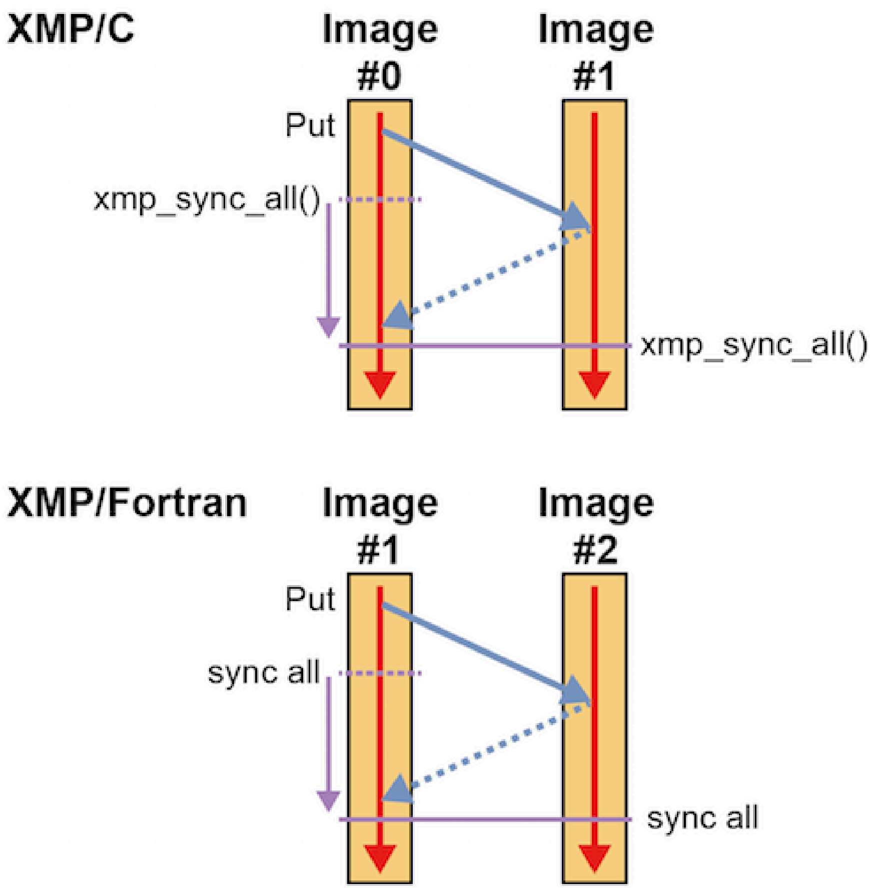

Fig. 51 `sync all`

XcalableMP Fortran
```
sync memory
```

Each image waits until all one-sided communication is complete. This function/statement does not imply barrier synchronization, unlike `sync all` and `sync images`, and therefore can be locally executed.

6 Procedure Interface

Procedure calls in XMP are almost the same as those in the base language. Procedure calls between other languages or to external libraries are also allowed if the base language supports them.

In the below example, a function/subroutine `sub1()` calls another function/subroutine `sub2()` with a **distributed array** `x` 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
```

To handle a parameter or dummy argument as a **global data** in the callee procedure, the programmer need to explicitly distribute it with an `align` directive.

If no `align` directive is specified in the callee procedure for a parameter or dummy argument that is declared as a **global data** in the caller procedure, it is handled as if it were declared in the callee procedure as a **local data** on each **node**, as follows.

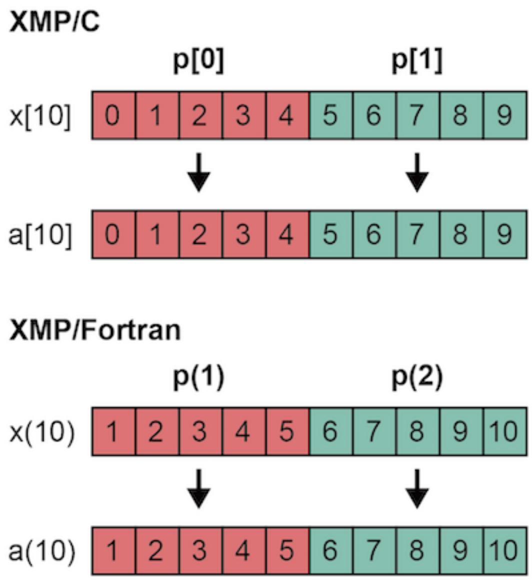

Fig. 52 Passing a global argument to a global parameter.

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
```

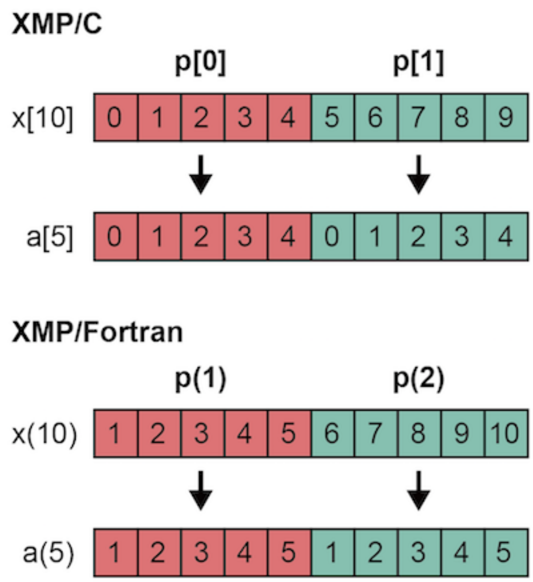

Fig. 53 Passing a global argument to a local parameter.

7 XMPT Tool Interface

7.1 Overview

XMPT is the tool interface of XMP and inspired by OMPT, which is the tool interface of OpenMP [4]. Hence, XMPT is designed as event-based and callback-based as OMPT; that is, for each event at runtime, the corresponding callback is invoked. One or more XMPT events are defined correspondig to each of XMP constructs and coarray-related actions (e.g. remote write/read and synchronization).

XMPT is preliminarily implemented in the Omni XMP compiler **??**, and used in MUST [5] and experimentally in Extrae [6]. More details of the application of XMPT in MUST is described in

.

7.2 Specification

7.2.1 Initialization

Tool developers can provide the `xmpt_initialize` function in which they register a callback for each of the XMPT events of their interset, as follows.

C

```
void xmpt_initialize(...){
 xmpt_set_callback(xmpt_event_bcast_begin, callback_bcast_begin);
 xmpt_set_callback(xmpt_event_bcast_end, callback_bcast_end);
 ...
}
```

In the above example, the tool developer implements callbacks `callback_bcast_begin` and `callback_bcast_end` that interact with his/her tool.

When an XMP program starts execution, the XMP runtime implicitly invokes `xmpt_initialize`, if provided, to set up the callbacks.

7.2.2 Events

XMPT defines XMPT events each of which corresponds to an XMP construct or a coarray-related action. Below is the list of XMPT events. For each of the events, the function signature of the corresponding callback is specifically defined. Note that the ones from `xmpt_event_coarray_remote_write` to `xmpt_event_sync_images_end` are coarray-related.

```
xmpt_event_task_begin
xmpt_event_task_end
xmpt_event_tasks_begin
xmpt_event_tasks_end
xmpt_event_loop_begin
xmpt_event_loop_end
xmpt_event_array_begin
xmpt_event_array_end
xmpt_event_reflect_begin
xmpt_event_reflect_begin_async
xmpt_event_reflect_end
xmpt_event_gmove_begin
xmpt_event_gmove_begin_async
xmpt_event_gmove_end
xmpt_event_barrier_begin
xmpt_event_barrier_end
xmpt_event_reduction_begin
xmpt_event_reduction_begin_async
xmpt_event_reduction_end
```

```
xmpt_event_bcast_begin
xmpt_event_bcast_begin_async
xmpt_event_bcast_end
xmpt_event_wait_async_begin
xmpt_event_wait_async_end
xmpt_event_coarray_remote_write
xmpt_event_coarray_remote_read
xmpt_event_coarray_local_write
xmpt_event_coarray_local_read
xmpt_event_sync_memory_begin
xmpt_event_sync_memory_end
xmpt_event_sync_all_begin
xmpt_event_sync_all_end
xmpt_event_sync_image_begin
xmpt_event_sync_image_end
xmpt_event_sync_images_all_begin
xmpt_event_sync_images_all_end
xmpt_event_sync_images_begin
xmpt_event_sync_images_end
```

When one of the XMPT events for which callbacks are registered occurs at runtime, the corresponding callback is invoked by the XMP runtime. For example, if callbacks are registered for events `xmpt_event_bcast_begin` and `xmpt_event_bcast_end` as in the example in the previous section, tha callbacks `callback_bcast_begin` and `callback_bcast_end` are invoked immediately before and after each of `bcast` constructs, respectively.

The XMP runtime passes therein all the information about the construct, including the mapping of the target global arrays, to the callback as its parameters. Thus, the tool is able to extract necessary information from the arguments.

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).
4. OpenMP Architecture Review Board, "OpenMP Application Programming Interface Version 5.0" (2018).
5. The MUST Project, https://www.itc.rwth-aachen.de/must.
6. The Extrae Project, https://tools.bsc.es/extrae.