XcalableACC: an Integration of XcalableMP and OpenACC

Akihiro Tabuchi and H. Murai, M. Nakao, T. Odajima, and T. Boku

Abstract XcalableACC (XACC) is an extension of XcalableMP for accelerated clusters. It is defined as a diagonal integration of XcalableMP and OpenACC, which is another directive-based language designed to program heterogeneous CPU/accelerator systems. XACC has features for handling distributed-memory parallelism, inherited from XMP, offloading tasks to accelerators, inherited from OpenACC, and two additional functions: data/work mapping among multiple accelerators and direct communication between accelerators.

Akihiro Tabuchi
Fujitsu Laboratories Ltd., 4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki, Kanagawa 211-8588, Japan, e-mail: tabuchi@fujitsu.com??????

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

Tetsuya Odajima
RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan, e-mail: tetsuya.odajima@riken.jp

Taisuke Boku
Center for Computationl Sciences, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan, e-mail: taisuke@ccs.tsukuba.ac.jp

Contents

1 Introduction

This document defines the specification of XcalableACC (XACC) which is an extension of XcalableMP version 1.3[1] and OpenACC version 2.5[2]. XcalableACC provides a parallel programming model for accelerated clusters which are distributed memory systems equipped with accelerators.

In this document, terminologies of XcalableMP and OpenACC are indicated by **bold font**. For details, refer to each specification[1, 2].

The works on XACC and the Omni XcalableACC compiler was supported by the Japan Science and Technology Agency, Core Research for Evolutional Science and Technology program entitled “Research and Development on Unified Environment of Accelerated Computing and Interconnection for Post-Petascale Era” in the research area of “Development of System Software Technologies for Post-Peta Scale High Performance Computing.”

1.1 Hardware Model

The target of XcalableACC is an accelerated cluster, a hardware model of which is shown in Fig. 1.

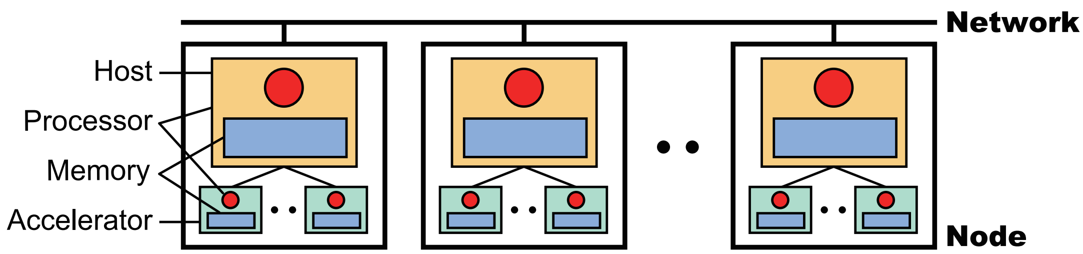

Fig. 1 Hardware Model

An execution unit is called **node** as with XcalableMP. Each **node** consists of a single host and multiple accelerators (such as GPUs and Intel MICs). Each host has a processor, which may have several cores, and own local memory. Each accelerator also has them. Each **node** is connected with each other via network. Each **node** can access its local memories directly and remote memories, that is, the memories of another **node** indirectly. In a host, the accelerator memory may be physically and/or virtually separate from the host memory as with the memory model of OpenACC. Thus, a host may not be able to read or write the accelerator memory directly.

1.2 Programming Model

XcalableACC is a directive-based language extension based on Fortran 90 and ISO C90 (ANSI C90). To develop applications on accelerated clusters with ease, XcalableACC extends XcalableACC and OpenACC independently as follow: (1) XcalableMP extensions are to facilitate cooperation between XcalableMP and OpenACC directives. (2) OpenACC extensions are to deal with multiple accelerators.

1.2.1 XcalableMP Extensions

In a program using the XcalableMP extensions, XcalableMP, OpenACC, and XcalableACC directives are used. Fig. 2 shows a concept of the XcalableMP extensions.

XcalableMP directives define a **template** and a **node set**. The **template** represents a global index space, which is distributed onto the **node set**. Moreover, XcalableMP directives declare **distributed arrays**, parallelize loop statements and transfer data among host memories according to the distributed **template**. OpenACC directives

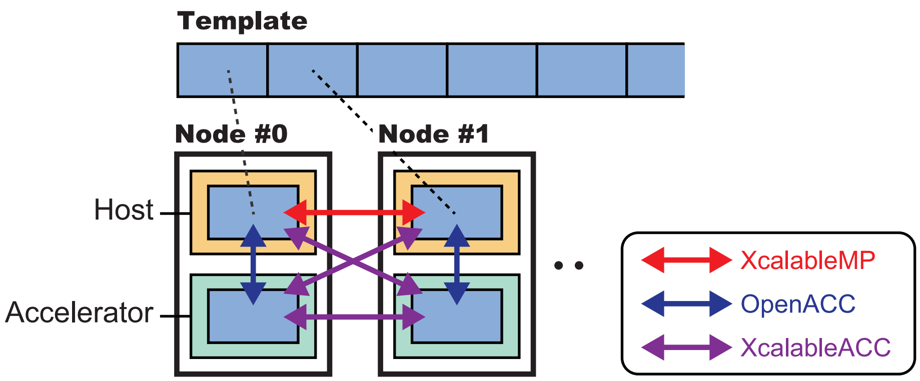

Fig. 2 Concept of XcalableMP Extensions

transfer the **distributed arrays** between host memory and accelerator memory on the same **node** and execute the loop statements parallelized by XcalableMP on accelerators in parallel. XcalableACC directives, which are XcalableMP communication directives with an `acc` clause, transfer data among accelerator memories and between accelerator memory and host memory on different **nodes**. Moreover, **coarray** features also transfer data on different nodes.

Note that the XcalableMP extensions are not a simple combination of XcalableMP and OpenACC. For example, if you represent communication of **distributed array** among accelerators shown in Fig. 2 by the combination of XcalableMP and OpenACC, you need to specify explicitly communication between host and accelerator by OpenACC and that between hosts by XcalableMP. Moreover, you need to calculate manually indices of the **distributed array** owned by each **node**. By contrast, XcalableACC directives can represent such communication among accelerators directly using global indices.

1.2.2 OpenACC Extensions

The OpenACC extension can represent offloading works and data to multiple-accelerators on a **node**. Fig. 3 shows a concept of the OpenACC extension.

OpenACC extension directive defines a **device set**. The **device set** represents a set of devices on a **node**. Futher, OpenACC extension directives declare **distributed arrays** on the **device set** while maintaining the arrays on the host memory, and the directives distribute offloading loop statement and memory copy between host and device memories for the **distributed-arrays**. Moreover, OpenACC extension directives synchronizes devices among the **device set**. XcalableACC directives also transfer data between device memories on the **node**.

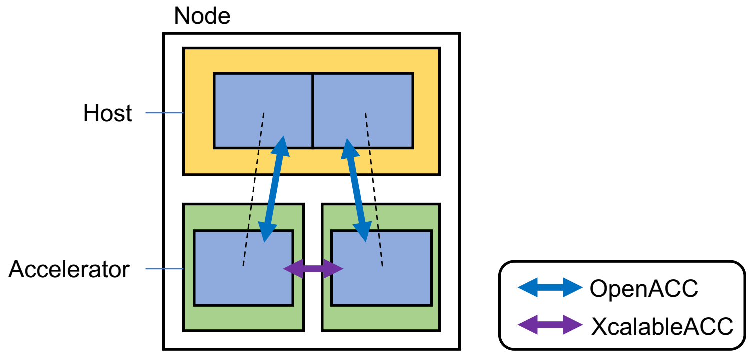

Fig. 3 Concept of OpenACC Extension

1.3 Execution Model

The execution model of XcalableACC is a combination of those of XcalableMP and OpenACC. While the execution model of a host CPU programming is based on that of XcalableMP, that of an accelerator programming is based on that of OpenACC. Unless otherwise specified, each **node** behaves exactly as specified in the XcalableMP specification[1] or the OpenACC specification[2].

An XcalableACC program execution is based on the SPMD model, where each **node** starts execution from the same main routine and keeps executing the same code independently (i.e. asynchronously), which is referred to as the replicated execution until it encounters an XcalableMP construct or an XcalableMP-extension construct. In particular, the XcalableMP-extension construct may allocate, deallocate, or transfer data on accelerators. An OpenACC construct or an OpenACC-extension construct may define **parallel regions**, such as work-sharing loops, and offloads it to accelerators under control of the host.

When a **node** encounters a loop construct targeted by a combination of XcalableMP `loop` and OpenACC `loop` directives, it executes the loop construct in parallel with other **accelerators**, so that each iteration of the loop construct is independently executed by the **accelerator** where a specified data element resides.

When a **node** encounters a XcalableACC synchronization or a XcalableACC communication directive, synchronization or communication occurs between it and other accelerators. That is, such **global constructs** are performed collectively by the **current executing nodes**. Note that neither synchronizations nor communications occur without these constructs specified.

1.4 Data Model

There are two classes of data in XcalableACC: **global data** and **local data** as with XcalableMP. Data declared in an XcalableACC program are local by default. Both **global data** and **local data** can exist on host memory and accelerator memory. About the data models of host memory and accelerator memory, refer to the OpenACC specification[2].

Global data are ones that are distributed onto the **executing node set** by the `align` directive. Each fragment of a **global data** is allocated in host memory of a **node** in the **executing node set**. OpenACC directives can transfer the fragment from host memory to accelerator memory.

Local data are all of the ones that are not global. They are replicated in the local memory of each of the **executing nodes**.

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

Particularly in XcalableACC Fortran, for common blocks that include any global variables, the ways how the storage sequence of them is defined and how the storage association of them is resolved are implementation-dependent.

1.5 Directive Format

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

`xxx` `type-face` characters are used to indicate literal type characters.
xxx... If the line is followed by "...", then xxx can be repeated.
[xxx] xxx is optional.
█ The syntax rule continues.
`[F]` The following lines are effective only in XcalableACC Fortran.
`[C]` The following lines are effective only in XcalableACC C.

In XcalableACC Fortran, XcalableMP and OpenACC directives are specified using special comments that are identified by unique sentinels `!$xmp` and `!$acc` respectively. the directives follow the rules for comment lines of either the Fortran free or fixed source form, depending on the source form of the surrounding program unit[1]. The directives are case-insensitive.

[1] Consequently, the rules of comment lines that an XcalableMP directive follows is the same as the ones that an OpenMP directive follows.

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

In XcalableACC, XcalableMP and OpenACC directives are specified using the `#pragma` mechanism provided by the C standards. the directives are case-sensitive.

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

1.6 Organization of This Document

The remainder of this document is structured as follows:

- Chapter 2: XcalableMP Extensions
- Chapter 3: OpenACC Extensions

2 XcalableMP Extensions

This chapter defines a behavior of mixing XcalableMP and OpenACC. Note that the existing OpenACC is not extended in the XcalableMP extensions. The XcalableMP extensions can represent (1) parallelization with keeping sequential code image using a combination of XcalableMP and OpenACC, and (2) communication among accelerator memories and between accelerator memory and host memory on different **nodes** using XcalableACC directives or **coarray** features.

2.1 Combination of XcalableMP and OpenACC

2.1.1 OpenACC Directives on Data

Description

When **distributed arrays** appear in OpenACC constructs, global indices in **distributed arrays** are used. The **distributed arrays** may appear in the `update`, `enter data`, `exit data`, `host_data`, `cache`, and `declare` directives, and the `data` clause accompanied by some of `deviceptr`, `present`, `copy`, `copyin`, `copyout`, `create`, and `delete` clauses. Data transfer of **distributed array** by OpenACC is performed on only **nodes** which have elements specified by the global indices.

Example

XcalableACC Fortran

```
integer :: a(N), b(N)
!$xmp template t(N)
!$xmp nodes p(*)
!$xmp distribute t(block) onto p
5 !$xmp align a(i) with t(i)
!$xmp align b(i) with t(i)
...
!$acc enter data copyin(a(1:K))
!$acc data copy(b)
10 ...
```

XcalableACC C

```
int a[N], b[N];
#pragma xmp template t[N]
#pragma xmp nodes p[*]
#pragma xmp distribute t[block] onto p
5 #pragma xmp align a[i] with t[i]
#pragma xmp align b[i] with t[i]
...
#pragma acc enter data copyin(a[0:K])
#pragma acc data copy(b)
10 { ...
```

Fig. 4 Code example in XcalableMP extensions with `enter_data` directive

In lines 2-6 of Fig. 4, the directives declare the **distributed arrays** *a* and *b*. In line 8, the `enter data` directive transfers the certain range of the **distributed array** *a* from host memory to accelerator memory. Note that the range is represented by global indices. In line 9, the `data` directive transfers the whole **distributed array** *b* from host memory to accelerator memory.

2.1.2 OpenACC Loop Construct

Description

In order to perform a loop statement on accelerators in **nodes** in parallel, XcalableMP `loop` directive and OpenACC `loop` directive are used. While XcalableMP `loop` directive performs a loop statement in **nodes** in parallel, OpenACC `loop` directive also performs the loop statement parallelized by the XcalableMP `loop` directive on accelerators in parallel. For ease of writing, the order of XcalableMP `loop` directive and OpenACC `loop` directive does not matter.

When `acc` clause appears in XcalableMP loop directive with `reduction` clause, the directive performs a reduction operation for a variable specified in the `reduction` clause on accelerator memory.

Restriction

- In OpenACC **compute region**, only XcalableMP `loop` directive without `reduction` clause can be inserted.
- In OpenACC **compute region**, targeted loop condition (lower bound, upper bound, and step of the loop) must remain unchanged.
- `acc` clause in XcalableMP loop directive can appear only when `reduction` clause appears there.

Example 1

In lines 2-6 of Fig. 5, the directives declare **distributed arrays** *a* and *b*. In line 8, the `parallel` directive with the `data` clause transfers the **distributed arrays** *a* and *b* from host memory to accelerator memory. Moreover, in lines 8-9, the `parallel` directive and XcalableMP `loop` directive perform the next loop statement on accelerators in **nodes** in parallel.

Example 2

In lines 2-5 of Fig. 6, the directives declare **distributed array** *a*. In line 7, the `parallel` directive with the `data` clause transfers the **distributed array** *a* and variable **sum** from host memory to accelerator memory. Moreover, in lines 7-8, the `parallel` directive and XcalableMP `loop` directive perform the next loop

XcalableACC Fortran

```
integer :: a(N), b(N)
!$xmp template t(N)
!$xmp nodes p(*)
!$xmp distribute t(block) onto p
!$xmp align a(i) with t(i)
!$xmp align b(i) with t(i)
...
!$acc parallel loop copy(a, b)
!$xmp loop on t(i)
do i=0, N
  b(i) = a(i)
end do
!$acc end parallel
```

XcalableACC C

```
int a[N], b[N];
#pragma xmp template t[N]
#pragma xmp nodes p[*]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i]
#pragma xmp align b[i] with t[i]
...
#pragma acc parallel loop copy(a, b)
#pragma xmp loop on t[i]
for(int i=0;i<N;i++){
  b[i] = a[i];
}
```

Fig. 5 Code example in XcalableMP extensions with OpenACC loop construct

XcalableACC Fortran

```
1  integer :: a(N), sum = 10
2  !$xmp template t(N)
3  !$xmp nodes p(*)
4  !$xmp distribute t(block) onto p
5  !$xmp align a(i) with t(i)
6  ...
7  !$acc parallel loop copy(a, sum) reduction(+:sum)
8  !$xmp loop on t(i) reduction(+:sum) acc
9  do i=0, N
10   sum = sum + a(i)
11 end do
12 !$acc end parallel loop
```

XcalableACC C

```
1  int a[N], sum = 10;
2  #pragma xmp template t[N]
3  #pragma xmp nodes p[*]
4  #pragma xmp distribute t[block] onto p
5  #pragma xmp align a[i] with t[i]
6  ...
7  #pragma acc parallel loop copy(a, sum) reduction(+:sum)
8  #pragma xmp loop on t[i] reduction(+:sum) acc
9  for(int i=0;i<N;i++){
10   sum += a[i];
11 }
```

Fig. 6 Code example in XcalableMP extensions with OpenACC loop construct with reduction clause

statement on accelerators in **nodes** in parallel. When finishing the calculation of the loop statement, OpenACC `reduction` clause and XcalableMP `reduction` and `acc` clauses in lines 7-8 perform a reduction operation for the variable **sum** on accelerators in **nodes**.

2.2 Communication on Accelerated Clusters

2.2.1 XcalableACC Directives

XcalableACC directives are extensions of `reflect`, `gmove`, `barrier`, `reduction`, `bcast`, and `wait_async` directives in XcalableMP global-view memory model. Moreover, `reflect_init` and `reflect_do` directives are added as extensions of the `reflect` directive. XcalableACC directives are directives which are added an `acc` clause to the above directives. XcalableACC directives transfer data stored on accelerator memory. Note that while XcalableACC `gmove` directive described in Section 2.2.1.1 and **coarray** features described in Section 2.2.2 can perform communication both among accelerator memories and between accelerator memory and host memory on different **nodes**, other directives can perform communication only among accelerator memories.

This section describes only the extended parts of XcalableACC directives from XcalableMP directives. For other information, refer to the XcalableMP specification[1].

2.2.1.1 reflect Construct

Synopsis

The `reflect` construct assigns the value of a reflection source to the corresponding shadow object.

Syntax

`[F] !$xmp reflect (` *array-name [*, *array-name]...* `)`
  [`width (` *reflect-width [*, *reflect-width]...* `)`*] [*`orthogonal`*] [*`async (` *async-id* `)`*] [*`acc`*]*
`[C] #pragma xmp reflect (` *array-name [*, *array-name]...* `)`
  [`width (` *reflect-width [*, *reflect-width]...* `)`*] [*`orthogonal`*] [*`async (` *async-id* `)`*] [*`acc`*]*

where *reflect-width* must be one of:

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

Description

When the `acc` clause is specified, the `reflect` construct updates each of the shadow object of the array specified by *array-name* on accelerator memory with the value of its corresponding reflection source.

Restriction

- When the `acc` clause is specified, the arrays specified by the sequence of *array-name*'s must be allocated on accelerator memory.
- This construct must not appear in OpenACC **compute region**.

Example

XcalableACC Fortran

```
integer :: a(N)
!$xmp template t(N)
!$xmp nodes p(*)
!$xmp distribute t(block) onto p
5 !$xmp align a(i) with t(i)
!$xmp shadow a(1)
...
!$acc enter data copyin(a)
!$xmp reflect (a) acc
```

XcalableACC C

```
int a[N];
#pragma xmp template t[N]
#pragma xmp nodes p[*]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i] 5
#pragma xmp shadow a[1]
...
#pragma acc enter data copyin(a)
#pragma xmp reflect (a) acc
```

Fig. 7 Code example in `reflect` construct

In lines 2-5 of Fig. 7, the directives declare **distributed array** *a*. In line 6, the `shadow` directive allocates shadow areas of the **distributed array** *a*. In line 8, the `enter data` directive transfers the **distributed array** *a* with the shadow areas from host memory to accelerator memory. In line 9, the `reflect` directive updates the shadow areas of the **distributed array** *a* on accelerator memory between neighboring **nodes**.

2.2.1.2 reflect_init and reflect_do Constructs

Synopsis

Since the `reflect_init` construct performs the initialization processes of the `reflect` construct, the `reflect_do` construct performs communication of the `reflect` construct.

Syntax

`[F] !$xmp reflect_init (` *array-name [, array-name]...* `)`
  [`width (` *reflect-width [, reflect-width]...* `)`*] [*`orthogonal`*] [*`async (` *async-id* `)`*] [*`acc`*]*
`[C] #pragma xmp reflect_init (` *array-name [, array-name]...* `)`
  [`width (` *reflect-width [, reflect-width]...* `)`*] [*`orthogonal`*] [*`async (` *async-id* `)`*] [*`acc`*]*

where *reflect-width* must be one of:

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

`[F] !$xmp reflect_do (` *array-name [, array-name]...* `)` *[*`async (` *async-id* `)`*] [*`acc`*]*
`[C] #pragma xmp reflect_do (` *array-name [, array-name]...* `)` *[*`async (` *async-id* `)`*] [*`acc`*]*

Description

The `reflect` construct is divided into `reflect_init` and `reflect_do` constructs to improve performance like the MPI persistent communication[3].

As a typical example, if a `reflect` construct is called repeatedly with the same condition in a loop statement, inserting a `reflect_init` construct before the loop statement and replacing the `reflect` construct with a `reflect_do` construct will improve its performance because unneeded initialization processes are removed.

Restriction

- When the `acc` clause is specified, the arrays specified by the sequence of *array-name*'s must be allocated on accelerator memory.
- These constructs must not appear in OpenACC **compute region**.
- The `reflect_init` directive must execute before the `reflect_init` directive executes.

Example

In lines 2-5 of Fig. 8, the directives declare **distributed array** *a*. In line 6, the `shadow` directive allocates shadow areas of the **distributed array** *a*. In line 8, the `enter data` directive transfers the **distributed array** *a* with the shadow areas from host memory to accelerator memory. In line 9, the `reflect_init` directive performs initialization processes for the `reflect_do` construct which targets the **distributed array** *a*. In line 11, the `reflect_do` directive updates the shadow areas of the

XcalableACC Fortran
```
integer :: a(N)
!$xmp template t(N)
!$xmp nodes p(*)
!$xmp distribute t(block) onto p
!$xmp align a(i) with t(i)
!$xmp shadow a(1)
...
!$acc enter data copyin(a)
!$xmp reflect_init (a) acc
...
!$xmp reflect_do (a) acc
```

XcalableACC C
```
int a[N];
#pragma xmp template t[N]
#pragma xmp nodes p[*]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i]
#pragma xmp shadow a[1]
...
#pragma acc enter data copyin(a)
#pragma xmp reflect_init (a) acc
...
#pragma xmp reflect_do (a) acc
```

Fig. 8 Code example in `reflect_init` and `reflect_do` constructs

distributed array *a* on accelerator memory between neighboring **nodes** without its initialization processes.

2.2.1.3 gmove Construct

Synopsis

The `gmove` construct allows an assignment statement, which may cause communication, to be executed possibly in parallel by the executing **nodes**.

Syntax

`[F] !$xmp gmove` *[*`in` | `out`*] [*`async (` *async-id* `)`*] [*`acc`*[*(*variable*)*]]*
`[C] #pragma xmp gmove` *[*`in` | `out`*] [*`async (` *async-id* `)`*] [*`acc`*[*(*variable*)*]]*

Description

- When the `acc` clause is specified and the variable is not specified by *variable* in the parenthesis, variables of both sides in the assignment statement on accelerator memory are targeted.
- When the `acc` clause is specified and the variable is specified by *variable* in the parenthesis, the specified variable on accelerator memory is targeted, and the unspecified variable on host memory is targeted.

Restriction

- The variables targeted on accelerator memory must be allocated on accelerator memory.
- This construct must not appear in OpenACC **compute region**.

Example

XcalableACC Fortran

```
integer :: a(N), b(N)
!$xmp template t(N)
!$xmp nodes p(*)
!$xmp distribute t(block) onto p
5 !$xmp align a(i) with t(i)
!$xmp align b(i) with t(i)
...
!$acc enter data copyin(a, b)
!$xmp gmove acc
10   a(:) = b(:)

!$xmp gmove acc(b)
  a(:) = b(:)
```

XcalableACC C

```
int a[N], b[N];
#pragma xmp template t[N]
#pragma xmp nodes p[*]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i] 5
#pragma xmp align b[i] with t[i]
...
#pragma acc enter data copyin(a, b)
#pragma xmp gmove acc
  a[:] = b[:]; 10

#pragma xmp gmove acc(b)
  a[:] = b[:];
```

Fig. 9 Code example in `gmove` construct

In lines 2-6 of Fig. 9, the directives declare **distributed arrays** *a* and *b*. In line 8, the `enter data` directive transfers the **distributed arrays** *a* and *b* from host memory to accelerator memory. In lines 9-10, the `gmove` construct copies the whole **distributed array** *b* to that of the **distributed array** *a* on accelerator memories. In lines 12-13, the `gmove` construct copies the whole **distributed array** *b* on accelerator memory to that of the **distributed array** *a* on host memory.

2.2.1.4 barrier Construct

Synopsis

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

Syntax

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

Description

- When the acc clause is specified, the barrier construct blocks until all ongoing asynchronous operations on accelerators are completed.
- When the acc clause is not specified, the barrier construct does not guarantee that an ongoing asynchronous operation on accelerator is completed.

Example

XcalableACC Fortran

```
!$xmp nodes p(*)
...
!$xmp barrier acc
```

XcalableACC C

```
#pragma xmp nodes p[*]
...
#pragma xmp barrier acc
```

Fig. 10 Code example in barrier construct

In line 1, the nodes directive defines node set *p*. In line 3, the barrier directive performs a barrier operation for accelerators on all **node**.

2.2.1.5 reduction Construct

Synopsis

The reduction construct performs a reduction operation among **nodes**.

Syntax

[F] !$xmp reduction (*reduction-kind : variable [*, *variable]...*)
*[*on *node-ref | template-ref] [*async (*async-id*)*] [*acc*]*

where *reduction-kind* is one of:

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

[C] `#pragma xmp reduction` (*reduction-kind : variable [*, *variable]...*)
[`on` *node-ref | template-ref] [*`async` (*async-id*)*] [*`acc`*]*

where *reduction-kind* is one of:

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

Description

When the `acc` clause is specified, the `reduction` construct performs a type of reduction operation specified by *reduction-kind* for the specified local variables among the accelerators and sets the reduction results to the variables on each of the accelerators.

Restriction

- When the `acc` clause is specified, the variables specified by the sequence of *variable*'s must be allocated on accelerator memory.
- This construct must not appear in OpenACC **compute region**.

Example

XcalableACC Fortran

```
integer :: a
!$xmp nodes p(*)
...
!$acc enter data copyin(a)
5 !$xmp reduction(+:a) acc
```

XcalableACC C

```
int a;
#pragma xmp nodes p[*]
...
#pragma acc enter data copyin(a)
5 #pragma xmp reduction(+:a) acc
```

Fig. 11 Code example in `reduction` construct

In line 2, the `nodes` directive defines **node set** *p*. In line 4, the `enter data` directive transfers the local variable *a* from host memory to accelerator memory. In line 5, the `reduction` directive calculates a total value of the variable *a* stored on each accelerator memory in each **node**.

2.2.1.6 bcast Construct

Synopsis

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

Syntax

`[F] !$xmp bcast (` *variable [*, *variable]...* `)` *[*`from` *nodes-ref | template-ref]* █
█ *[*`on` *nodes-ref] | template-ref] [*`async (` *async-id* `)`*] [*`acc`*]*
`[C] #pragma xmp bcast (` *variable [*, *variable]...* `)` *[*`from` *nodes-ref | template-ref]* █
█ *[*`on` *nodes-ref | template-ref] [*`async (` *async-id* `)`*] [*`acc`*]*

Description

When the `acc` clause is specified, the values of the variables specified by the sequence of *variable*'s on accelerator memory (called **broadcast variables**) are broadcasted from the **node** specified by the `from` clause (called the **source node**) to each of the **nodes** in the **node set** specified by the `on` clause. After executing this construct, the values of the **broadcast variables** become the same as those in the **source node**.

Restriction

- When the `acc` clause is specified, the variables specified by the sequence of *variable*'s must be allocated on accelerator memory.
- This construct must not appear in OpenACC **compute region**.

Example

XcalableACC Fortran

```
integer :: a
!$xmp nodes p(*)
...
!$acc enter data copyin(a)
5 !$xmp bcast(a) acc
```

XcalableACC C

```
int a;
#pragma xmp nodes p[*]
...
#pragma acc enter data copyin(a)
#pragma xmp bcast(a) acc 5
```

Fig. 12 Code example in `bcast` construct

In line 2, the `nodes` directive defines **node set** *p*. In line 4, the `enter data` directive transfers the local variable *a* from host memory to accelerator memory. In line 5, the `bcast` directive broadcasts the variable *a* stored on accelerator memory to all *nodes*.

2.2.1.7 wait_async Construct

Synopsis

The `wait_async` construct guarantees asynchronous communications specified by *async-id* are complete.

Syntax

`[F] !$xmp` `wait_async ( `*async-id* *[* `,` *async-id* *]*`...)` *[*`on` *nodes-ref* | *template-ref]* *[*`acc`*]*
`[C] #pragma xmp` `wait_async ( `*async-id* *[* `,` *async-id* *]*`...)` *[*`on` *nodes-ref* | *template-ref]*
[`acc`*]*

Description

When the `acc` clause is specified, the `wait_async` construct blocks and therefore statements following it are not executed until all of the asynchronous communications that are specified by *async-id*'s and issued on the accelerators in **node set** specified by the `on` clause are complete.

Restriction

This construct must not appear in OpenACC **compute region**.

Example

XcalableACC Fortran

```fortran
integer :: a
!$xmp nodes p(*)
...
!$acc enter data copyin(a)
5 !$xmp reduction(+:a) acc async(1)
...
!$xmp wait_async(1) acc
```

XcalableACC C

```c
int a;
#pragma xmp nodes p[*]
...
#pragma acc enter data copyin(a)
5 #pragma xmp reduction(+:a) acc async(1)
...
#pragma xmp wait_async(1) acc
```

Fig. 13 Code example in `wait_async` construct

In line 2, the `nodes` directive defines **node set** *p*. In line 4, the `enter data` directive transfers the local variable *a* from host memory to accelerator memory. In line 5, the `reduction` directive performs asynchronously. In line 7, the `wait_async` construct blocks until the asynchronous reduction operation at line 5 is complete.

2.2.2 Coarray Features

Synopsis

XcalableACC can perform one-sided communication (put/get operations) for data on accelerator memory using **coarray** features, which is based on XcalableMP local-view memory model. A combination of **coarray** syntax and `host_data` construct enables communication between accelerators.

Description

If **coarrays** appear in `use_device` clause of any enclosing `host_data` construct, communication targets data on the accelerator side. **Coarray** operations on accelerators are synchronized using the same synchronization functions in XcalableMP.

Restriction

- Only `declare` directive can declare a **coarray** on accelerator memory. For example, `enter data` and `copy` directives cannot declare a **coarray** on accelerator memory.
- The **coarray** syntax must not appear in OpenACC **compute region**.

Example

XcalableACC Fortran

```
1  integer :: a(N)[*]
2  integer :: b(N)
3  !$acc declare create(a, b)
4  ...
5  if(this_image() == 1) then
6  !$acc host_data use_device(a, b)
7    a(:)[2] = b(:)
8  
9  !$acc host_data use_device(a)
10   b(:) = a(:)[3]
11 end if
12 ...
13 sync all
```

XcalableACC C

```
1  int a[N]:[*];
2  int b[N];
3  #pragma acc declare create(a, b)
4  ...
5  if(xmp_node_num() == 1){
6  #pragma acc host_data use_device(a, b)
7    a[:]:[2] = b[:];
8  
9  #pragma acc host_data use_device(a)
10   b[:] = a[:]:[3];
11 }
12 ...
13 xmp_sync_all(NULL);
```

Fig. 14 Code example in coarray features

In line 3 of Fig. 14, the `declare` directive declares a **coarray** *a* and an array *b* on accelerator memory. In lines 6-7, **node** 1 performs put operation, where the whole array *b* on accelerator memory in **node** 1 is transferred to the **coarray** *a* on accelerator memory in **node** 2. In lines 9-10, **node** 1 performs get operation, where the whole **coarray** *a* on accelerator memory in **node** 3 is transferred to the array *b* on host memory in **node** 1. In line 13, the `sync all` statement in XcalableACC Fortran or the `xmp_sync_all` function in XcalableACC C synchronizes all **nodes** and guarantees completion of ongoing coarray operations.

3 OpenACC Extensions

This chapter defines an extension of OpenACC in XcalableACC. The extension can represent offload works to multiple-accelerators on each **node**.

3.1 Device Set Definition and Reference

3.1.1 devices Directive

Synopsis

The **devices** directive declares a set of devices.

Syntax

`[F] !$acc devices` *devices-decl [*, *devices-decl]...*
`[C] #pragma acc devices` *devices-decl [*, *devices-decl]...*

where *device-decl* is one of:

devices-name `(` *devices-spec* `)`
devices-name `(` *devices-spec* `)` *[= predefined-devices-ref]*
`[C]` *devices-name* `[` *devices-spec* `]`
`[C]` *devices-name* `[` *devices-spec* `]` *[= predefined-devices-ref]*

and *devices-spec* is one of:

*
int-expr

and *predefined-devices-ref* is one of:

device-type-name `(` * | *int-expr* | *int-expr* : *int-expr*`)`
device-type-name `[` * | *int-expr* | *int-expr* : *int-expr*`]`

Description

The `device` directive declares a device array that corresponds to a device set.

The first and third forms are used to declare a device array that corresponds to a set of the entire default devices. The second and fourth forms are used to declare a device array, each device of which is assigned to a device of the device set is specified by *predefined-devices-ref* at the corresponding position.

Restriction

- *devices-name* must not conflict with any other local name in the same scoping unit.
- This construct must not appear in OpenACC **compute region**.

Example

The following are examples of the devices declaration. The device array *d* corresponds to a set of entire default devices and the device array *e* is a subset of the predefined device array *nvidia*. The program must be executed by a node which has four or more NVIDIA accelerator devices.

XcalableACC Fortran

```
!$acc devices d(*)
!$acc devices e(2) = nvidia(3:4)
```

XcalableACC C

```
#pragma acc devices d[*]
#pragma acc devices e[2] = nvidia[2:2]
```

Fig. 15 Code example in XcalableACC `devices` directive

3.1.1.1 Default Device Set

Synopsis

The default device set is the targeting device set when the `on_device` clause is omitted.

Description

The default device set is the device set which contains the all OpenACC default devices on the node. The device type of each device of the set equals to *acc_device_default*, and the size of the set equals to a result of *acc_get_num_devices(acc_device_default)*.

3.1.1.2 Device Reference

Synopsis

The device reference is used to reference a device set.

Syntax

devices-ref **is** *devices-name [* `(` *devices-subscript* `)` *]*
`[C]` *devices-ref* **is** *devices-name [* `[` *devices-subscript* `]` *]*

where *devices-subscript* must be one of:

int-expr
triplet

Description

A device reference by *devices-name* represents a device set corresponding to the device array specified by the name or its subarray.

Example

Assume that *d* is the name of a device array.

- To specify a device set to which the declared device array corresponds,

XcalableACC Fortran

```
!$acc devices e(1) = d(1)
!$acc devices f(3) = d(2:4)
```

XcalableACC C

```
#pragma acc devices e[1] = d[0]
#pragma acc devices f[3] = d[1:3]
```

- To specify a device array that corresponds to the executing device array set in the `barrier` directive.

XcalableACC Fortran

```
!$acc barrier_device on_device(d)
```

XcalableACC C

```
#pragma acc barrier_device on_device(d)
```

3.1.2 on_device clause

Synopsis

The `on_device` clause specifies a execution device set for the directive.

Syntax

`on_device(` *devices-ref* `)`

Description

The `on_device` clause may appear on `parallel`, `parallel loop`, `kernels`, `kernels loop`, `data`, `enter data`, `exit data`, `declare`, `update`, `wait`, and `barrier_device` directives.

The `on_device` clause specifies a device set which the directive targets. The directive is applied to each device of the device set in parallel. If there is no `layout` clause, the all devices process the directive for same data or work redundantly.

If no `on_device` clause appears on a `declare` directive with a `layout` clause, it is assumed that the default device set is specified by `on_device` clause. If no `on_device` clause appears on a `barrier_device` directive, it is assumed that the default device set is specified by `on_device` clause. If no `on_device` clause appears on a `data`, `enter data`, `exit data`, or `update` directives, if the arrays are already declared by `declare` directive, the device set that specified at the `declare` directive is targeted. In the other cases, the directive behaves the same as normal OpenACC.

3.2 Data and Work Mapping Clauses

3.2.1 layout Clause

Synopsis

The `layout` clause specifies data or work mapping on devices.

Syntax

In `declare` directive:

`[F] layout( (` *dist-format [*, *dist-format] ...* `) )`
`[C] layout( [` *dist-format* `]` *[* `[` *dist-format* `]` *] ...* `)`

where *dist-format* must be one of:

`*`
`block`

In `loop`, `parallel loop`, and `kernels loop` construct:

`[F] layout(` *array-name* `(` *layout-subscript [*, *layout-subscript] ...* `) )`
`[C] layout(` *array-name* `[` *layout-subscript* `]` *[* `[` *layout-subscript* `]` *] ...* `)`

where *layout-subscript* must be one of:

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

Description

The `layout` clause may appear on `declare` directives and on `loop`, `parallel loop`, and `kernels loop` constructs. If the `layout` clause appears on a `declare` directive, it specifies the data mapping to the device set for arrays which are appeared in data clauses on the directive. "*" represents that the dimension is not distributed, and `block` represents that the dimension is divided into contiguous blocks, which are distributed onto the device array.

If the `layout` clause appears on a `loop`, `parallel loop`, or `kernels loop` directive, it specifies the mapping for the immediately following loop. If *loop-index* appears in *layout-subscript*, the loop is distributed to the device set in the same manner as the dimension where the *loop-index* appears. If there is no `on_device` clause on the construct, it is assumed that the device set on which the array is distributed is specified by `on_device` clause.

Restriction

- *loop-index* must be a control variable of a loop.

Example

The following are examples of the layout clause. In line 2, the `devices` directive defines device set *d*. In line 3-4, the `declare` directive declares that an array *a* is distributed and allocated on the device set *d*. In line 6–9, the `kernels loop` directive distributes the loop and offloads the loops to the device set *d*.

3.2.2 shadow Clause

Synopsis

The `shadow` clause allocates the shadow area for a distributed array on devices.

XcalableACC Fortran

```
integer :: a(N)
!$acc devices d(*)
!$acc declare create(a)
!$acc+layout((block)) on_device(d)
...
!$acc kernels loop layout(a(i))
do i = 1, N
  a(i) = i * 2
end do
```

XcalableACC C

```
int a[N];
#pragma acc devices d[*]
#pragma acc declare create(a) \
        layout([block]) on_device(d)
...
#pragma acc kernels loop layout(a[i])
for(int i = 0; i < N; i++){
  a[i] = i * 2;
}
```

Fig. 16 Code example in XcalableACC `layout` clause

Syntax

`[F] shadow(` `(` *shadow-width [*, *shadow-width] ...* `)` `)`
`[C] shadow(` `[` *shadow-width* `]` *[* `[` *shadow-width* `]` *] ...* `)`

where *shadow-width* must be one of:

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

Description

The `shadow` clause may appear on `declare` directives. The `shadow` clause specifies the width of the shadow area of arrays on the `declare` directive, which is used to communicate the neighbor element of the block of the arrays. When *shadow-width* is of the form "*int-expr* : *int-expr*," the shadow area of the width specified by the first *int-expr* is added at the lower bound and that specified by the second one at the upper bound in the dimension. When *shadow-width* is of the form *int-expr*, the shadow area of the same width specified is added at both the upper and lower bounds in the dimension.

Restriction

- `shadow` clause must appear with `layout` clause.
- The value specified by *shadow-width* must be a non-negative integer.
- The number of *shadow-width* must be equal to the number of dimensions (or rank) of the arrays on the `declare` directive.

- If an array is also distributed on **nodes**, a *shadow-width* of `shadow` clause must be same as the *shadow-width* of XcalableMP `shadow` directive for the same dimension.

Example

The following are examples of the shadow clause. In line 2, the `devices` directive defines device set *d*. In line 3-5, the `declare` directive declares that an array *a* is distributed and allocated with shadow areas on the device set *d*. In line 7–10, the `kernels loop` construct divides and offloads the loop to the device set *d*. In line 11, the `reflect` directive updates the shadow areas of the distributed array *a* on devices.

XcalableACC Fortran

```
1  integer :: a(N)
2  !$acc devices d(*)
3  !$acc declare create(a)
4  !$acc+layout((block))
5  !$acc+shadow((1:1)) on_device(d)
6  ...
7  !$acc kernels loop layout(a(i))
8  do i = 1, N
9    a(i) = i * 3
10 end do
11 !$acc reflect(a)
```

XcalableACC C

```
1  int a[N];
2  #pragma acc devices d[*]
3  #pragma acc declare create(a) \
4          layout([block]) \
5          shadow([1:1]) on_device(d)
6  ...
7  #pragma acc kernels loop layout(a[i])
8  for(int i = 0; i < N; i++){
9    a[i] = i * 3;
10 }
11 #pragma acc reflect(a)
```

Fig. 17 Code example in XcalableACC `shadow` clause

3.3 Synchronization on Accelerators

3.3.1 barrier_device Construct

Synopsis

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

Syntax

[F] `!$acc barrier_device` *[*`on_device(` *devices-ref* `)`*]*
[C] `#pragma acc barrier_device` *[*`on_device(` *devices-ref* `)`*]*

Description

The barrier_device construct blocks accelerator devices until all ongoing asynchronous operations on them are completed regardless of the host operations. The construct is performed among the device set specified by the `on_device` clause. If no `on_device` clause is specified, then it is assumed that the default device set is specified in it.

Restriction

- This construct must not appear in OpenACC **compute region**.

Example

The following are examples of the barrier_devices construct. In line 1–2, the `devices` directives define device set *d* and *e*. In line 4–5, the first `barrier_device` construct performs a barrier operation for all devices, and the second one performs a barrier operation for devices in the device set *e*.

XcalableACC Fortran

```
!$acc devices d(*)
!$acc devices e(2) = d(1:2)
...
!$acc barrier_device
5 !$acc barrier_device on_device(e)
```

XcalableACC C

```
#pragma acc devices d[*]
#pragma acc devices e[2] = d[0:2]
...
#pragma acc barrier_device
#pragma acc barrier_device on_device(e) 5
```

Fig. 18 Code example in XcalableACC `barrier_device` construct

References

1. XcalableMP Language Specification, http://xcalablemp.org/specification.html (2017).
2. The OpenACC Application Programming Interface, http://www.openacc.org (2015).
3. MPI: A Message-Passing Interface Standard, http://mpi-forum.org (2015).