XcalableACC

⟨*ex-scalable-a-c-c*⟩

Language Specification

Version 1.0

RIKEN AICS and University of Tsukuba

March, 2017

History

Version 1.0: March 31, 2017 First release.

Contents

Chapter 1

Introduction

This document defines the specification of XcalableACC 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 `typewriter font`. For details, refer to each specification[1, 2].

1.1 Hardware Model

The target of XcalableACC is an accelerated cluster, a hardware model of which is shown in Fig. 1.1.

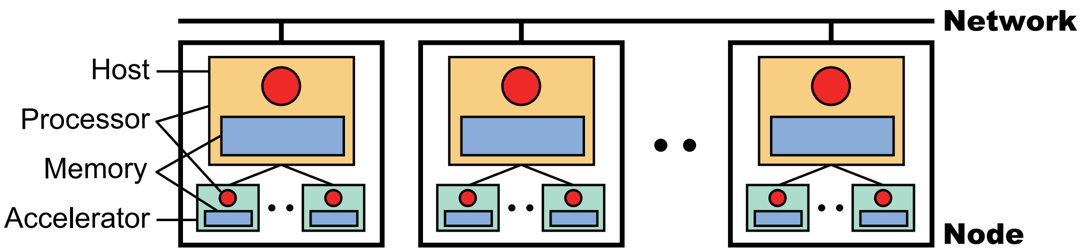

Figure 1.1: Hardware Model

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 Overview of XcalableACC

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 XcalableMP and OpenACC independently as follow: (1) XcalableMP extension is to facilitate cooperation between existing XcalableMP and OpenACC directives. (2) OpenACC extension is to deal with multiple accelerators.

1.2.1 XcalableMP Extension

In a program using the XcalableMP extension, XcalableMP, OpenACC, and XcalableACC directives are used. Fig. 1.2 shows a concept of the XcalableMP extension.

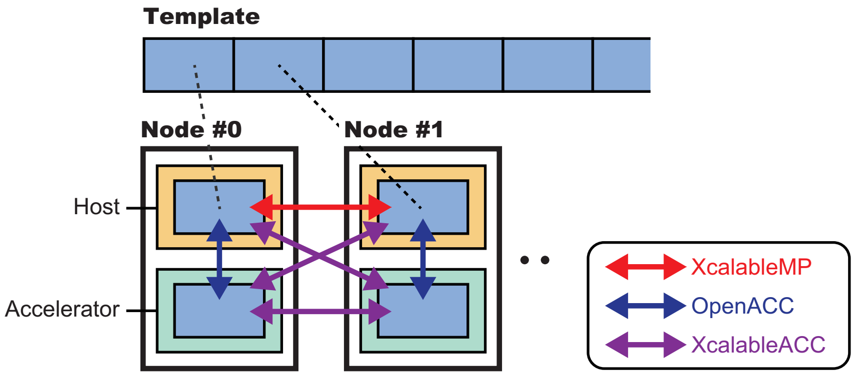

Figure 1.2: Concept of XcalableMP Extension

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 `template`. OpenACC directives transfer the `distributed arrays` between host memory and accelerator memory on the same `node` and calculate the loop statements parallelized by XcalableMP on accelerators. 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 among them.

The XcalableMP extension is defined to develop parallel applications with keeping the sequential code image. Note that the XcalableMP extension is not a simple combination of XcalableMP and OpenACC. For example, if you represent communication of `distributed array` among accelerators shown in Fig. 1.2 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 global indices of the `distributed array` owned by each `node`.

1.2.2 OpenACC Extensions

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 on XcalableACC is based on that of XcalableMP, that of an accelerator programming is based on that of OpenACC.

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 `distributed array` 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.

1.4 Organization of This Document

The remainder of this document is structured as follows:

- Chapter 2: XcalableMP Extension
- Chapter 3: OpenACC Extension

Chapter 2

XcalableMP Extension

This chapter defines a behavior of mixing XcalableMP and OpenACC. Note that the existing OpenACC is not extended in the XcalableMP extension. The XcalableMP extension 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` are appeared in OpenACC constructs, global indices in `distributed arrays` are used. `Distributed arrays` may be appeared in the OpenACC **update**, **enter data**, **exit data**, **host_data**, **cache**, and **declare** directives. Moreover, they may be appeared in 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 global indices.

Example

XcalableACC Fortran

```
1  integer :: a(N), b(N)
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  !$xmp align b(i) with t(i)
7  ...
8  !$acc enter data copyin(a(1:k))
9  !$acc data copy(b)
10 ...
```

XcalableACC C

```
1  int a[N], b[N];
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  #pragma xmp align b[i] with t[i]
7  ...
8  #pragma acc enter data copyin(a[0:k])
9  #pragma acc data copy(b)
10 { ...
```

Figure 2.1: Example of a code in XcalableMP extension with OpenACC data clause

In lines 2-6, XcalableMP directives declare the `distributed arrays` *a* and *b*. In line 8, the OpenACC **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 OpenACC **data** directive transfers the whole `distributed array` *b* from host memory to accelerator memory.

2.1.2 OpenACC Loop Construct

Description

In order to parallelize a loop statement on multiple accelerators on multiple `nodes`, XcalableMP **loop** directive and OpenACC **loop** directive are used. While XcalableMP **loop** directive parallelizes a loop statement on each `node`, OpenACC **loop** directive parallelizes the loop statement parallelized by XcalableMP **loop** directive on each accelerator. The order of XcalableMP **loop** directive and OpenACC **loop** directive does not matter.

Example

XcalableACC Fortran

```
integer :: a(N), b(N), sum = 0
!$xmp template t(N)
!$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 parallel copy(a, b, sum)

10 !$xmp loop on t(i)
!$acc loop
do i=0, N
  b(i) = a(i)
end do
15 !$xmp loop on t(i) reduction(+:sum)
!$acc loop reduction(+:sum)
do i=0, N
  sum = sum + b(i)
end do
20 !$acc end parallel
```

XcalableACC C

```
int a[N], b[N], sum = 0;
#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 parallel copy(a, b, sum)
{
#pragma xmp loop on t[i] 10
#pragma acc loop
  for(int i=0;i<N;i++){
    b[i] = a[i];
  }
#pragma xmp loop on t[i] reduction(+:sum) 15
#pragma acc loop reduction(+:sum)
  for(int i=0;i<N;i++){
    sum += b[i];
  }
} 20
```

Figure 2.2: Example of a code in XcalableMP extension with OpenACC loop construct

In lines 2-6, XcalableMP directives declare `distributed arrays` *a* and *b*. In line 8, OpenACC **parallel** directive starts `parallel region` and transfers the `distributed arrays` *a* and *b* and local variable *sum* from host memory to accelerator memory. In line 10, XcalableMP **loop** directive parallelizes the next loop statement depending on the `template` *t* on each `node`. In line 11, OpenACC **loop** directive also parallelizes the next loop statement parallelized by XcalableMP on each accelerator. In lines 15-16, **reduction** clauses are added in both **loop** directives. At the end of the both **loop** constructs, the reduction operations occur to calculate the total value of the local variable *sum* stored on each accelerator memory in each `node`.

2.2 Communication on Accelerated Clusters

2.2.1 XcalableACC Directives

XcalableACC directives are extensions of XcalableMP **reflect**, **gmove**, **barrier**, **reduction**, **bcast**, and **wait_async** directives in XcalableMP global-view memory model. When adding an **acc** clause to the above XcalableMP directives, data stored on accelerator memory are transferred shown in Fig. 1.2. Note that while XcalableACC **gmove** directive described in Section 2.2.1.1 and `coarray` features described in Section 2.2.2 can occur communication both among accelerator memories and between accelerator memory and host memory on different `nodes`, other directives can occur 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.

Example

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

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 (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 (a) acc
```

Figure 2.3: Example of a code in XcalableACC **reflect** construct

2.2.1.2 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.

Example

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

2.2.1.3 barrier Construct

Synopsis

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

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 enter data copyin(a, b)
!$xmp gmove acc
  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]
#pragma xmp align b[i] with t[i]
...
#pragma acc enter data copyin(a, b)
#pragma xmp gmove acc
  a[:] = b[:];

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

Figure 2.4: Example of a code in XcalableACC **gmove** construct

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 outgoing asynchronous operations on all accelerators are completed.
- When the **acc** clause is not specified, the barrier construct does not guarantee that an outgoing 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
```

Figure 2.5: Example of a code in XcalableACC **barrier** construct

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

2.2.1.4 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.

Example

XcalableACC Fortran

```
1 integer :: a
2 !$xmp nodes p(*)
3 ...
4 !$acc enter data copyin(a)
5 !$xmp reduction(+:a) acc
```

XcalableACC C

```
1 int a;
2 #pragma xmp nodes p[*]
3 ...
4 #pragma acc enter data copyin(a)
5 #pragma xmp reduction(+:a) acc
```

Figure 2.6: Example of a code in XcalableACC **reduction** construct

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

2.2.1.5 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.

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)
5 #pragma xmp bcast(a) acc
```

Figure 2.7: Example of a code in XcalableACC **bcast** construct

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

2.2.1.6 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.

Example

XcalableACC 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
```
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
```

Figure 2.8: Example of a code in XcalableACC **wait_async** construct

In line 2, XcalableMP **nodes** directive defines `node set` p. In line 4, OpenACC **enter data** directive transfers the local variable a from host memory to accelerator memory. In line 5, XcalableACC **reduction** directive calculates a total value of the variable a stored on accelerator memory in all `node` asynchronously. In line 7, XcalableACC **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 OpenACC **host_data** construct enables communication between accelerators.

Description

If `coarrays` appear in OpenACC **use_device** clause of any OpenACC 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 OpenACC **declare** directive can declare a `coarray` on accelerator memory. For example, OpenACC **enter data** and **copy** directives cannot declare a `coarray` on accelerator memory.

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);
```

Figure 2.9: Example of a code in XcalableACC coarray features

Example

In line 3, OpenACC **declare** directive declares a `coarray` *a* and a 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.

Chapter 3

OpenACC Extension

Acknowledgment

The work 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.”

Bibliography

[1] XcalableMP Specification, `http://xcalablemp.org/specification.html` (2017).

[2] OpenACC, `http://www.openacc.org` (2015).