Overview of XcalableMP

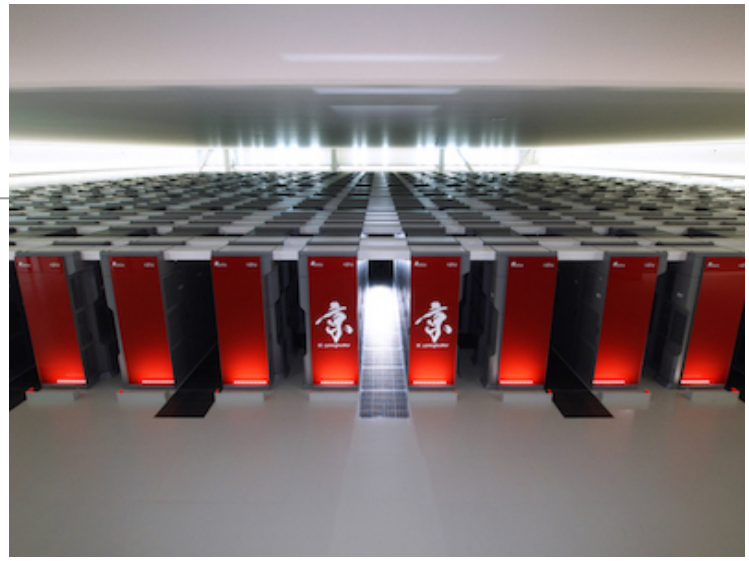
Masahiro Nakao (RIKEN AICS, Japan)

Agenda in the morning session

- Overview
- XcalableMP language
- Omni XcalableMP compiler
- How to install Omni XcalableMP compiler (Hands-on)
  - Create Hello World program, and execute it

Please feel free to interrupt me at any time if you have any questions.

Background



- Distributed memory systems are widely used for large-scale simulations, and so on.
- Message Passing Interface (MPI) is a de-facto standard for programming on these systems
- MPI programming is a **very** hard work.
  - MPI requires numerous code changes from a serial code.
  - It is necessary to divide data and calculations manually among compute nodes.

New programming language that could provide both high performance and high productivity has been demanded.

XcalableMP (XMP)

- Directive-based parallel language for C and Fortran
  - Now XMP/C++ on the table
  - Proposed by XMP Specification Working Group of PC Cluster Consortium
    - This Working Group consists of members from
      - Academia: U. Tsukuba, U. Tokyo, Kyoto U. and Kyusyu U.
      - Research labs: RIKEN, NIFS, JAXA, JAMSTEC/ES
      - Industries: Fujitsu, NEC, Hitachi
- The specification is available at http://xcalablemp.org

XcalableMP (XMP)

XMP/C

```
int a[100];
#pragma xmp nodes p[*]
#pragma xmp template t[100]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i]

#pragma xmp loop on t[i] reduction(+:res)
for(int i=0;i<100;i++){
  a[i] = i;
  res += a[i];
}
```

XMP/Fortran

```
integer :: a(100)
!$xmp nodes p(*)
!$xmp template t(100)
!$xmp distribute t(block) onto p
!$xmp align a(i) with t(i)

!$xmp loop on t(i) reduction(+:res)
do i=0, 100
  a(i) = i
  res = res + a(i)
end do
```

The same directives can be used in both languages.

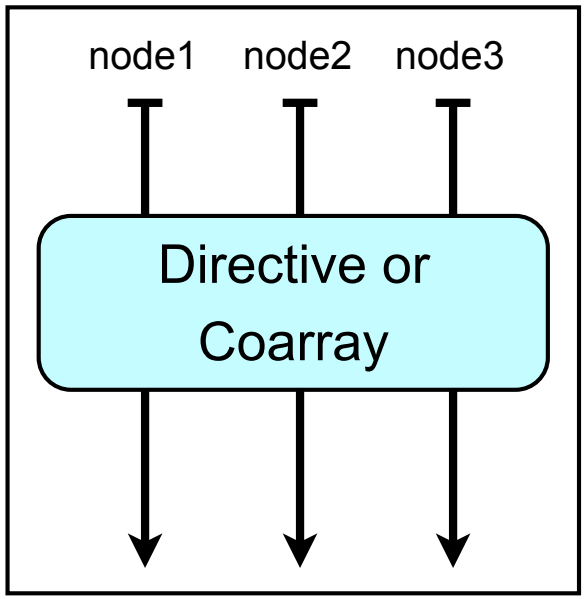
Features of XMP (1/2)

1. Directive-based language extension based on C and Fortran like OpenMP
   - Add XMP directives to a serial code
   - To reduce code-writing and educational costs
   - To reuse existing a serial code easily
2. Collaboration with MPI
   - To call **an MPI program from an XMP program** or
     to call **an XMP program from an MPI program**,
     XMP provides MPI programming interfaces.
   - This feature is to reuse existing an MPI code to develop new XMP applications easily.

Features of XMP (2/2)

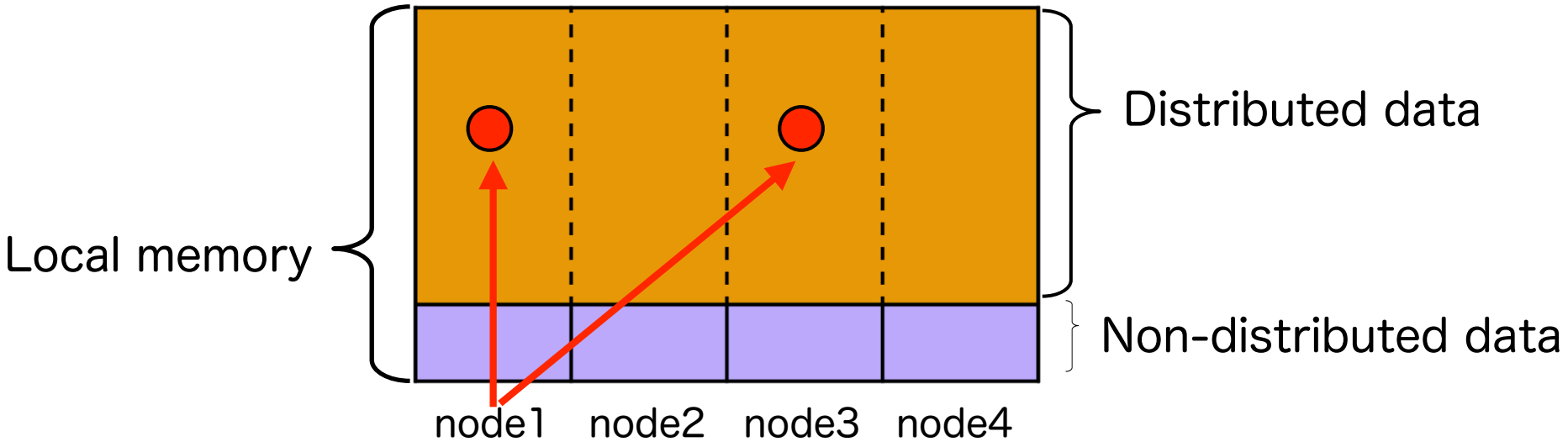
3. Global-view / Local-view memory models
   - Global-view memory model for typical parallelization **using directives**
   - Local-view memory model for one-sided communication **using coarray**

4. Performance-aware for explicit communication, synchronization and work-mapping
   - The basic execution model of XMP is SPMD
   - Execution unit in XMP is called "node"
   - Each node executes in parallel independently
   - All actions occur when directive or coarray is encountered for being “easy-to-understand” in performance tuning



Basic memory model

- Each node can directly access data on its own local memory.
- To access data on remote nodes, special constructs are needed.
  - Directives for global-view
  - Coarray for local-view
- Distributed data, which can be access by another node, is defined by directives or coarray features
- Non-distributed data are replicated on all node



XcalableMP (XMP)

XMP/C

```
int a[100], b[100];
#pragma xmp nodes p[*]
#pragma xmp template t[100]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i]
```

XMP/Fortran

```
integer :: a(100), b(100)
!$xmp nodes p(*)
!$xmp template t(100)
!$xmp distribute t(block) onto p
!$xmp align a(i) with t(i)
```

The array a(100) is distributed by XMP directives.

But, the array b(100) is not distributed, which is replicated on all node.

Global/Local-view memory models

- **Global-view memory model**
  - Programmer describes data/work mapping and communication using directives
  - Support typical patterns for data/work mapping and communication
  - Indices of arrays are global and distributed among nodes
- **Local-view memory model**
  - One-sided communication using coarray
  - Coarray of Fortran 2008
    - Intel, Cray, Fujitsu compilers support coarray features in Fortran
  - We also defines coarray features in C language as a part of XMP
  - Coarray communication is more flexible than XMP directive
  - Indices of arrays are local on each node

Agenda in the morning session

- Overview
- XcalableMP language
- Omni XcalableMP compiler
- How to install Omni XcalableMP compiler (Hands-on)
  - Create Hello World program

Agenda in the morning session

- Overview
- XcalableMP language
  - Global-view
    - Declare distributed array
    - Parallelize loop statement
    - Perform communication
  - Local-view
- Omni XcalableMP compiler
- How to install Omni XcalableMP compiler (Hands-on)
  - Create Hello World program

XMP directive rules

- XMP/C uses the **#pragma** mechanism.
- XMP/Fortran uses **comment lines**.
- Examples:

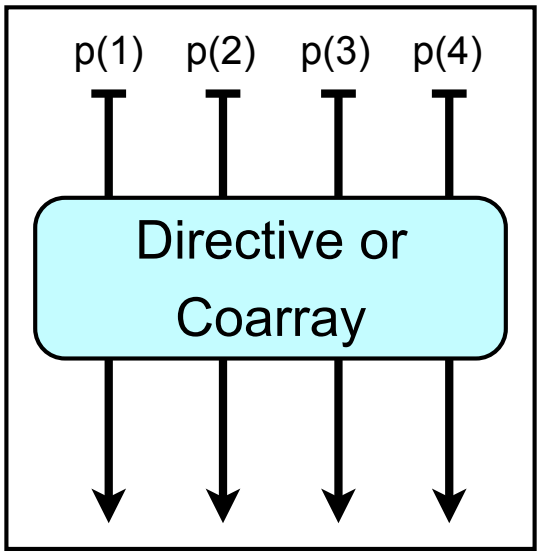
The node directive defines node set.

[C] `#pragma xmp nodes p[4]` p[0], p[1], p[2], p[3] are defined

[F] `!$xmp nodes p(4)` p(1), p(2), p(3), p(4) are defined

Subscript in square bracket is zero-origin.

Subscript in round bracket is one-origin.



Node directive

- Declare node array which is an execution unit set.
- Declare shape and size of the node array
- Examples

[C]

```
#pragma xmp nodes p[4]
#pragma xmp nodes p[2][4]
#pragma xmp nodes p[*]
#pragma xmp nodes p[*][4]
```

[F]

```
!$xmp nodes p(4)
!$xmp nodes p(4,2)
!$xmp nodes p(*)
!$xmp nodes p(4,*)
```

Four nodes run

Eight nodes run

must be a multiple of 4

The order of square bracket is based on C's (row order)

p[0][0], p[0][1], p[0][2], p[0][3], p[1][0], p[1][1], p[1][2], p[1][3]

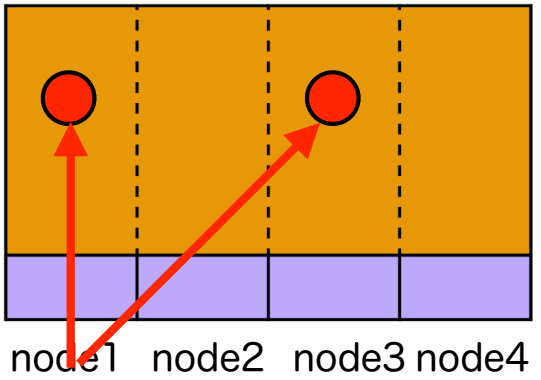
The order of round bracket is based on Fortran's (column order)

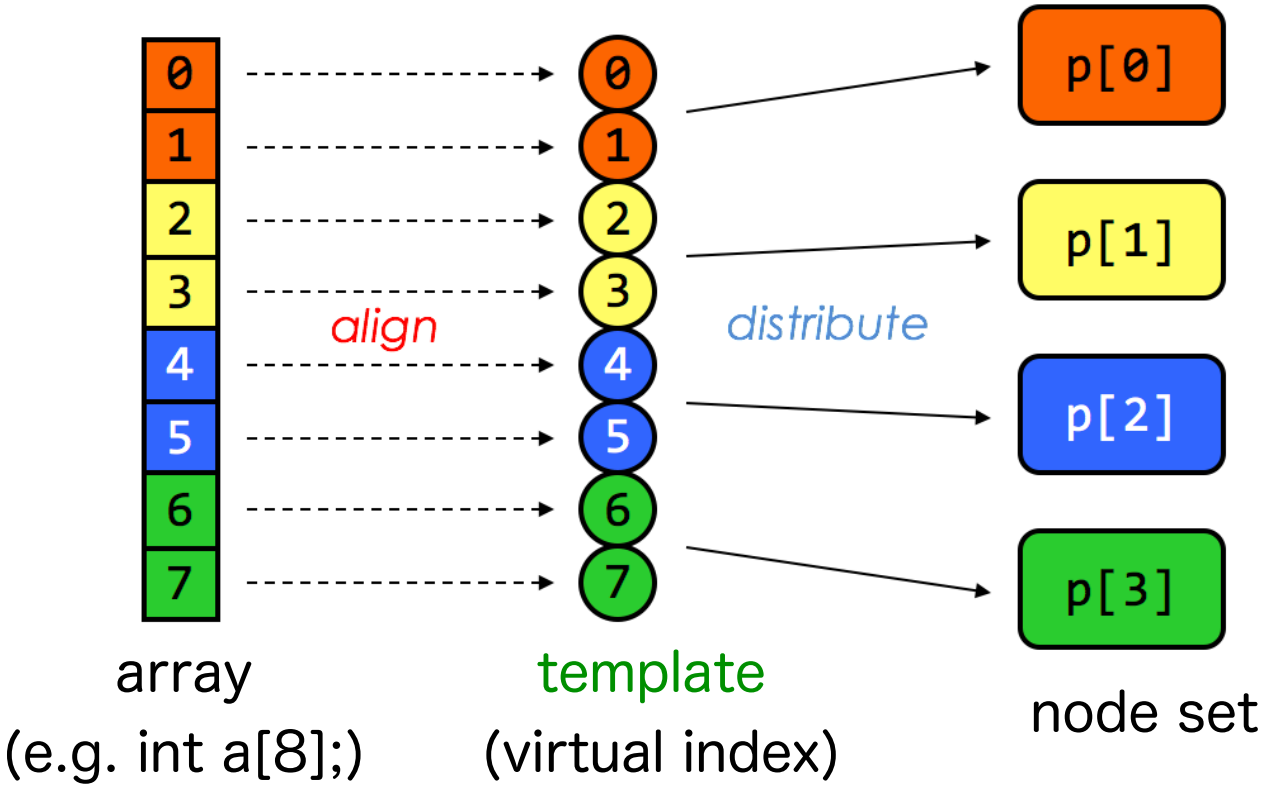
p(1,1), p(2,1), p(3,1), p(4,1), p(1,2), p(2,2), p(3,2), p(4,2)

The "*" represents the size of the node set is automatically adjusted according to the total size of process.

Distributed data

- How to declare distributed data.
- Two-level data mapping with *alignment* and *distribution*
  - 1.nodes, 2.template, 3.align, 4.distribute directives are used



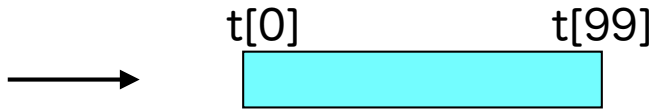


- Arrays are aligned with a template.
- The template is distributed onto nodes.

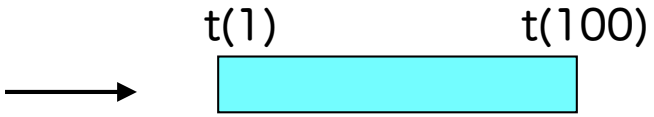
Template directive

- Declares the shape of a template, which is a virtual array as an index space
- A template is used as the target of data and work alignments
- Examples

[C] `#pragma xmp template t[100]`



[F] `!$xmp template t(100)`

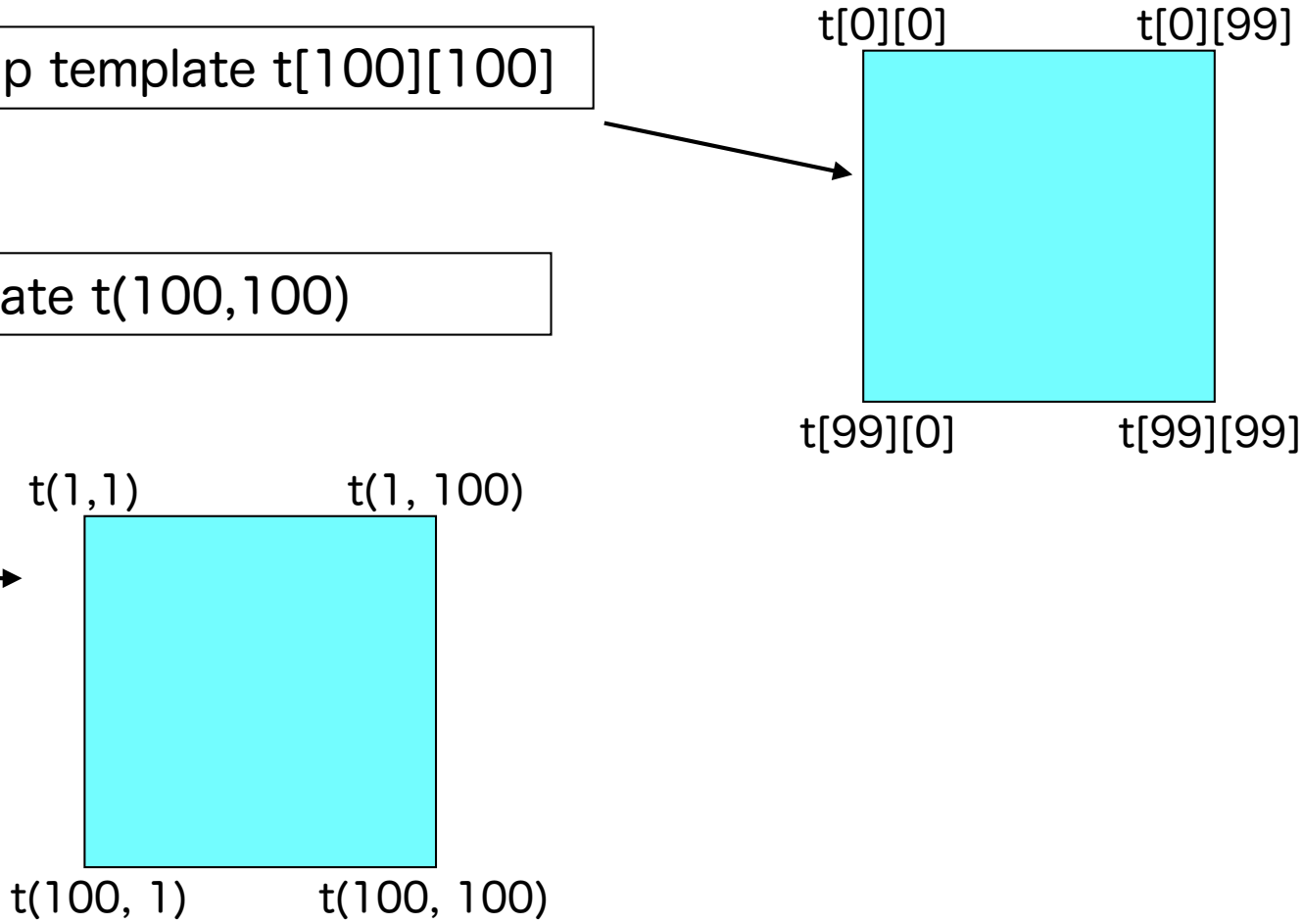


Template directive

- Declares the shape of a template, which is a virtual array (i.e. an index space)
- A template is used as the target of data and work alignments
- Examples

[C] `#pragma xmp template t[100][100]`

[F] `!$xmp template t(100,100)`



Distribute directive

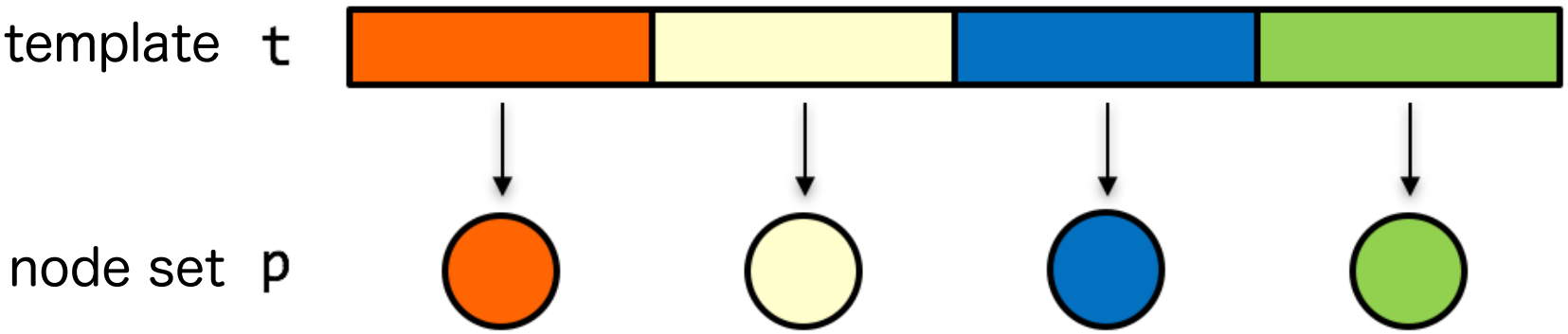
- Distributes a template onto a node array in the specified distribution format.
- Examples

[C]

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

[F]

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



Support cyclic, block-cyclic, and non-uniform block ("gblock") can also be specified as the distribution format.

Distribute directive

- Example (http://xcalablemp.org/datamapping.html)

block distribution

C

```
#pragma xmp nodes p[4]
#pragma xmp template t[20]
#pragma xmp distribute t[block] onto p
```

Fortran

```
!$xmp nodes p(4)
!$xmp template t(20)
!$xmp distribute t(block) onto p
```

node	indexes of template
p[0]	0, 1, 2, 3, 4
p[1]	5, 6, 7, 8, 9
p[2]	10, 11, 12, 13, 14
p[3]	15, 16, 17, 18, 19

node	indexes of template
p(1)	1, 2, 3, 4, 5
p(2)	6, 7, 8, 9, 10
p(3)	11, 12, 13, 14, 15
p(4)	16, 17, 18, 19, 20

Distribute directive

- Example (http://xcalablemp.org/datamapping.html)

cyclic distribution

C

```
#pragma xmp nodes p[4]
#pragma xmp template t[20]
#pragma xmp distribute t[cyclic] onto p
```

Fortran

```
!$xmp nodes p(4)
!$xmp template t(20)
!$xmp distribute t(cyclic) onto p
```

node	indexes of template
p[0]	0, 4, 8, 12, 16
p[1]	1, 5, 9, 13, 17
p[2]	2, 6, 10, 14, 18
p[3]	3, 7, 11, 15, 19

node	indexes of template
p(1)	1, 5, 9, 13, 17
p(2)	2, 6, 10, 14, 18
p(3)	3, 7, 11, 15, 19
p(4)	4, 8, 12, 16, 20

Distribute directive

- Example (http://xcalablemp.org/datamapping.html)

block-cyclic distribution

C

```
#pragma xmp nodes p[4]
#pragma xmp template t[20]
#pragma xmp distribute t[cyclic(2)] onto p
```

Fortran

```
!$xmp nodes p(4)
!$xmp template t(20)
!$xmp distribute t(cyclic(2)) onto p
```

node	indexes of template
p[0]	0, 1, 8, 9, 16, 17
p[1]	2, 3, 10, 11, 18, 19
p[2]	4, 5, 12, 13
p[3]	6, 7, 14, 15

node	indexes of template
p(1)	1, 2, 9, 10, 17, 18
p(2)	3, 4, 11, 12, 19, 20
p(3)	5, 6, 13, 14
p(4)	7, 8, 15, 16

Distribute directive

- Example (http://xcalablemp.org/datamapping.html)

gblock(m) distribution non-uniform block

C

```
#pragma xmp nodes p[4]
#pragma xmp template t[20]
int m[4] = {3, 5, 8, 4};
#pragma xmp distribute t[gblock(m)] onto p
```

Fortran

```
!$xmp nodes p(4)
!$xmp template t(20)
integer :: m(4) = (/3, 5, 8, 4/)
!$xmp distribute t(gblock(m)) onto p
```

node	indexes of template
p[0]	0, 1, 2
p[1]	3, 4, 5, 6, 7
p[2]	8, 9, 10, 11, 12, 13, 14, 15
p[3]	16, 17, 18, 19

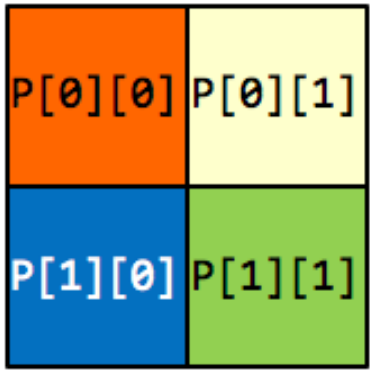
node	indexes of template
p(1)	1, 2, 3
p(2)	4, 5, 6, 7, 8
p(3)	9, 10, 11, 12, 13, 14, 15, 16
p(4)	17, 18, 19, 20

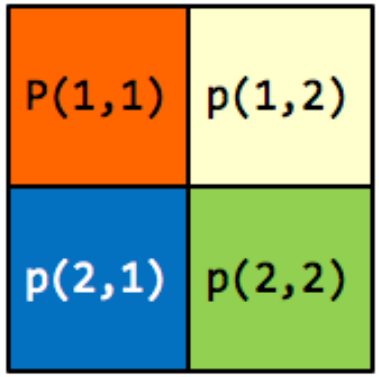
Distribute directive

- Examples for multi-dimensional template and node set

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

```
!$xmp nodes p(2,2)                                 [F]
!$xmp template t(10,10)
!$xmp distribute t(block,block) onto p
```





node	1st indexes of a[][]	2nd indexes of a[][]
p[0][0]	0, 1, 2, 3, 4	0, 1, 2, 3, 4
p[0][1]	0, 1, 2, 3, 4	5, 6, 7, 8, 9
p[1][0]	5, 6, 7, 8, 9	0, 1, 2, 3, 4
p[1][1]	5, 6, 7, 8, 9	5, 6, 7, 8, 9

node	1st indexes of a()	2nd indexes of a()
p(1,1)	1, 2, 3, 4, 5	1, 2, 3, 4, 5
p(2,1)	6, 7, 8, 9, 10	1, 2, 3, 4, 5
p(1,2)	1, 2, 3, 4, 5	6, 7, 8, 9, 10
p(2,2)	6, 7, 8, 9, 10	6, 7, 8, 9, 10

Align directive

- Aligns each element of an array with the specified element of a template.
- Examples

[C]

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

[F]

```
integer :: a(8)
!$xmp align a(i) with t[i]
```

Align the element i of an array a[] with the element i of a template t.

[C]

```
int a[10][10];
#pragma xmp align a[i][j] with t[i][j]
```

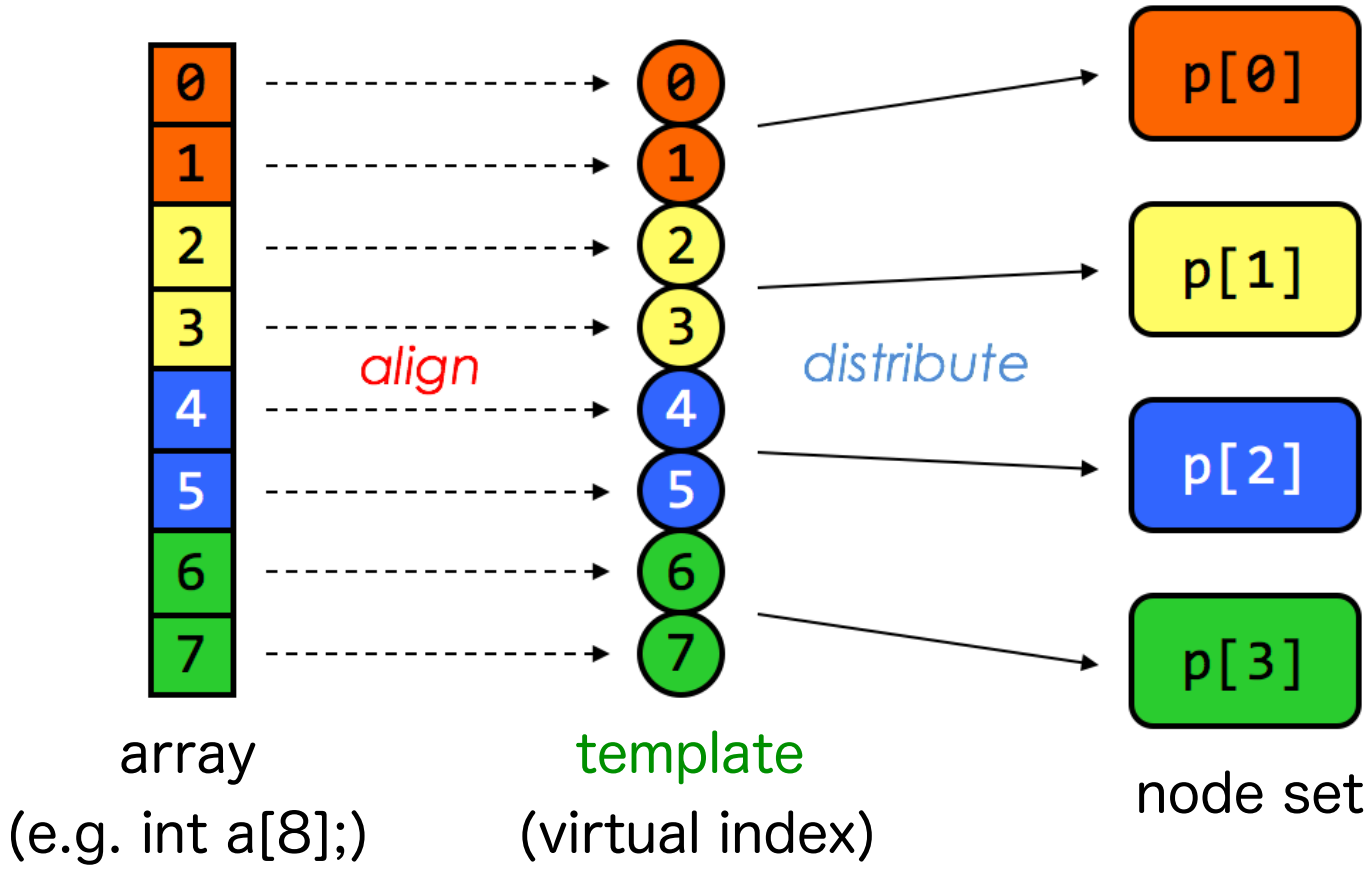
[F]

```
integer :: a(10,10)
!$xmp align a(i,j) with t[i][j]
```

Distributed data

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

- How to declare data on distributed data a
- Two-level data mapping with *alignment* and *distribution*
  - node, template, align. distribute directives are used

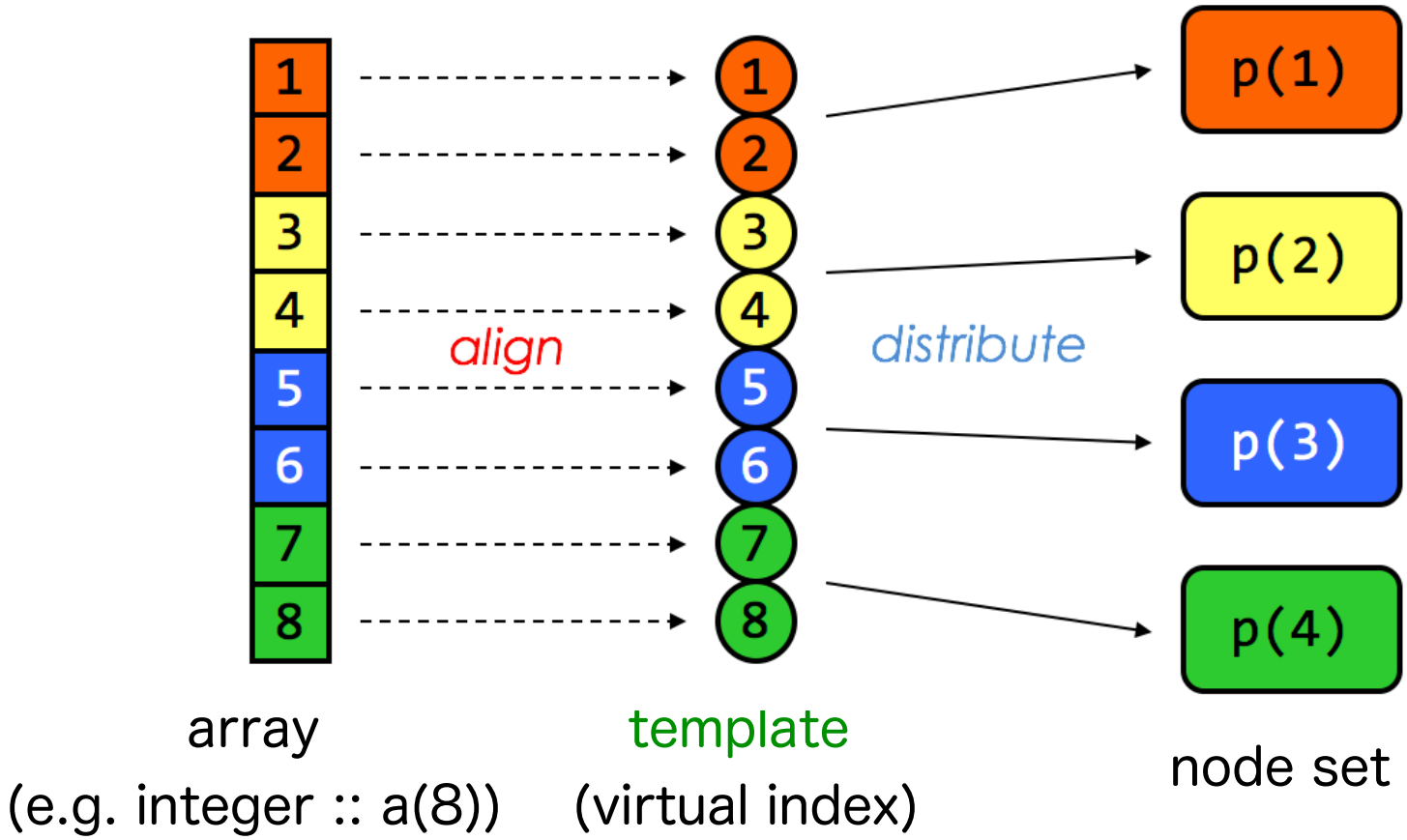


- Arrays are aligned with a template.
- The template is distributed onto nodes.

Distributed data

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

- How to declare data on distributed data a
- Two-level data mapping with *alignment* and *distribution*
  - node, template, align. distribute directives are used



- Arrays are aligned with a template.
- The template is distributed onto nodes.

Agenda in the morning session

- Overview
- XcalableMP language
  - Global-view
    - Declare distributed array
    - Parallelize loop statement
    - Perform communication
  - Local-view
- Omni XcalableMP compiler
- How to install Omni XcalableMP compiler (Hands-on)
  - Create Hello World program

Loop directive

- Parallelizes a following loop.
  - Specifies which node executes each iteration of the loop
    by "aligning" each iteration with an element of a template.
  - An iteration "i" is to be executed by the owner node of template t[i]

[C]

```
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t[block] onto p
int a[16];
#pragma xmp align a[i] with t[i]

#pragma xmp loop on t[i]
for(int i=0;i<16;i++){
    a[i] = func(i);
}
```

[F]

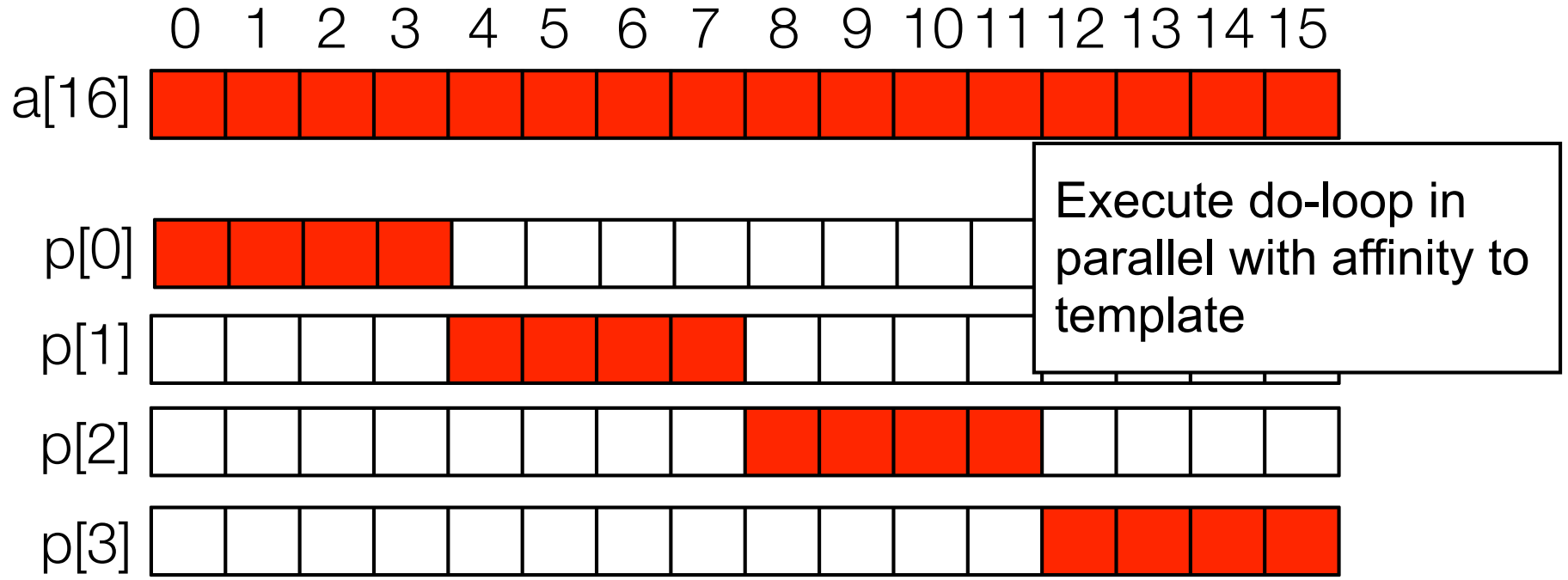
```
!$xmp nodes p(4)
!$xmp template t(16)
!$xmp distribute t(block) onto p
integer :: a(16)
!$xmp align a(i) with t(i)

!$xmp loop on t(i)
do i=1, 16
  a(i) = func(i)
end do
```

Loop directive in XMP/C

- Loop directive is inserted before do-loop

```
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i]
```

```
#pragma xmp loop on t(i)
for(int i=0;i<16;i++){
```



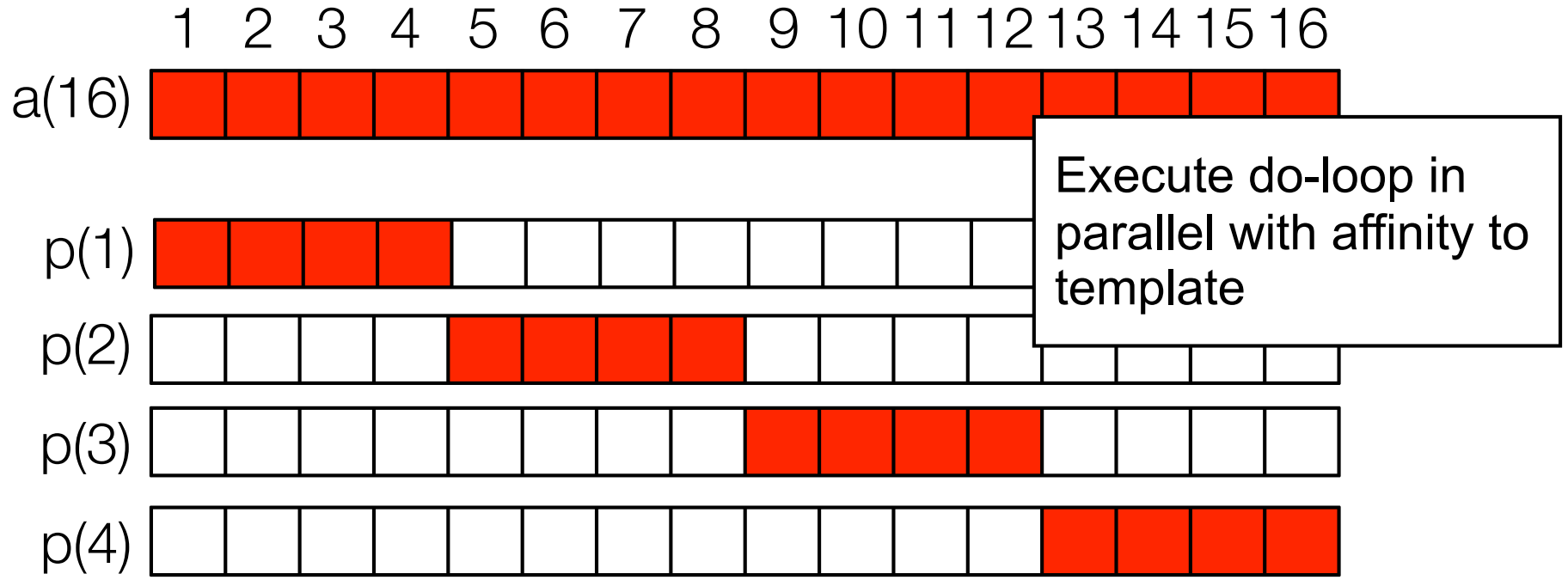
Each node computes **Red elements** in parallel

Loop directive in XMP/Fortran

- Loop directive is inserted before do-loop

```
!$xmp nodes p(4)
!$xmp template t(16)
!$xmp distribute t(block) onto p
!$xmp align a(i) with t(i)
```

```
!$xmp loop on t(i)
do i=1, 16
```



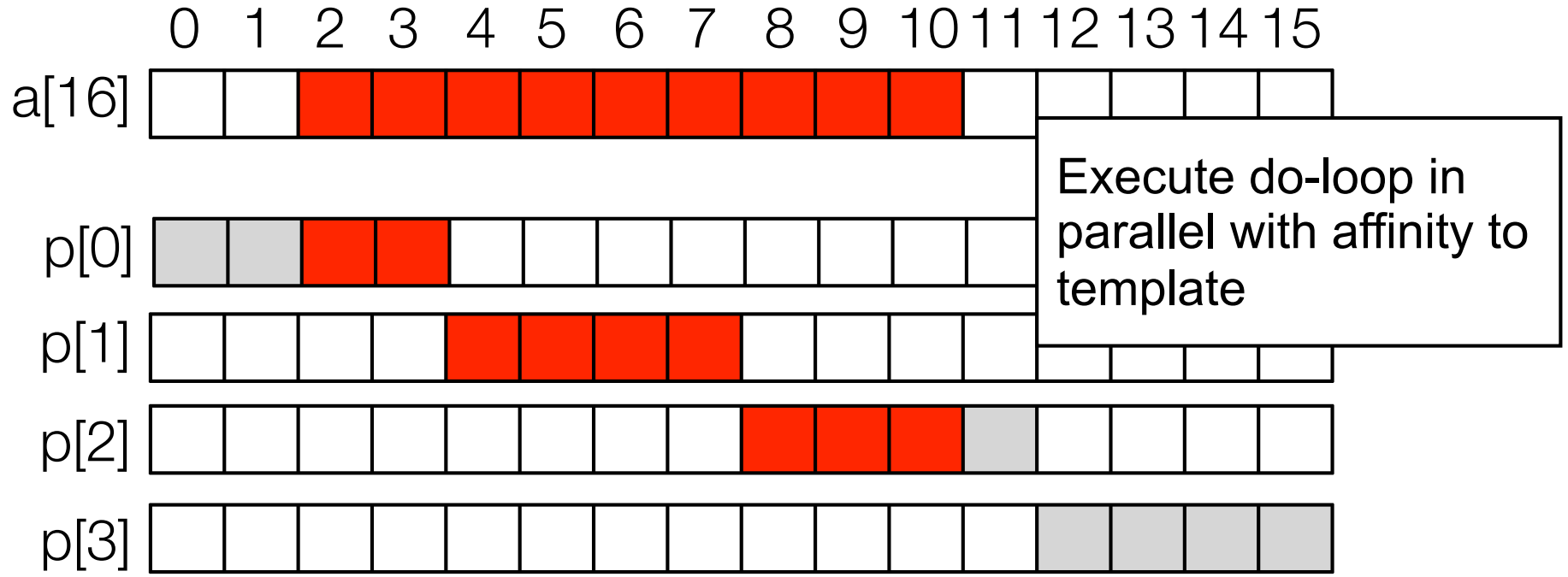
Each node computes **Red elements** in parallel

Loop directive in XMP/C

- Loop directive is inserted before do-loop

```
#pragma xmp nodes p[4]
#pragma xmp template t[16]
#pragma xmp distribute t[block] onto p
#pragma xmp align a[i] with t[i]
```

```
#pragma xmp loop on t(i)
for(int i=2;i<11;i++){
```



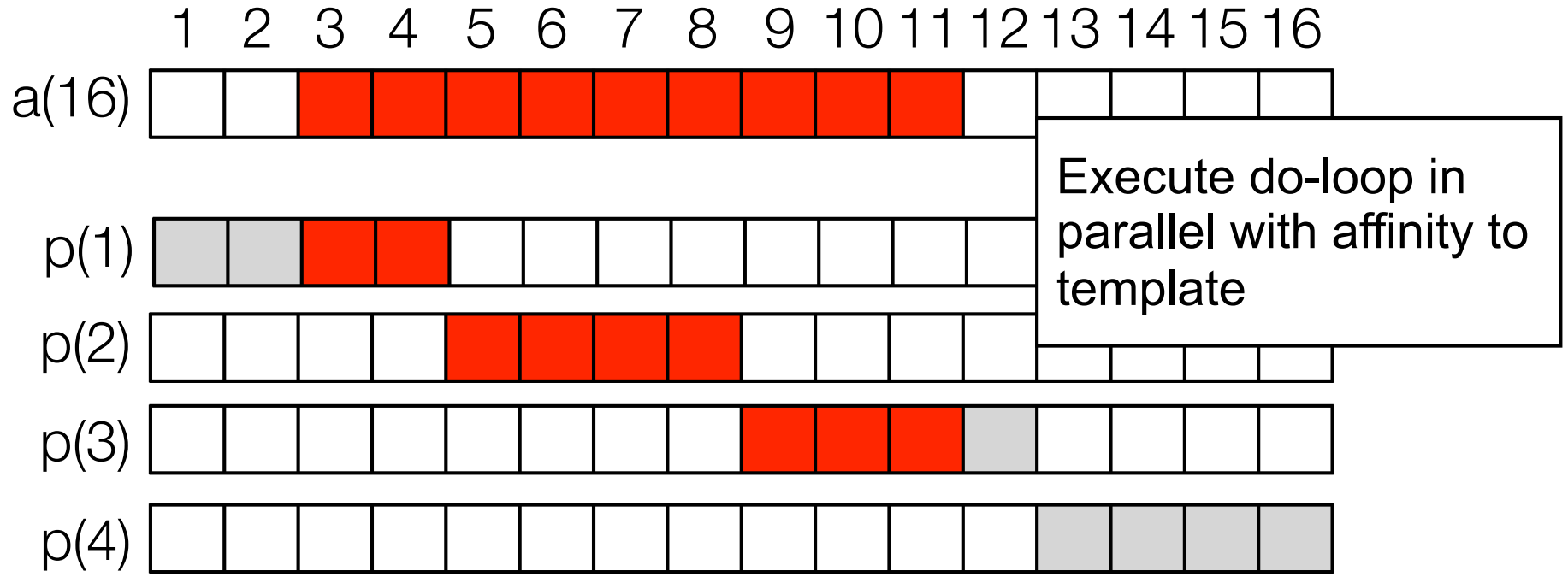
Each node computes **Red elements** in parallel

Loop directive in XMP/Fortran

- Loop directive is inserted before do-loop

```
!$xmp nodes p(4)
!$xmp template t(16)
!$xmp distribute t(block) onto p
!$xmp align a(i) with t(i)
```

```
!$xmp loop on t(i)
do i=3, 11
```



Each node computes **Red elements** in parallel

Loop directive

- Parallelizes the following loop(s).
  - Information of index is needed for a **nested loop** between “loop” and “on”

[C]

```
#pragma xmp nodes p[4][2]
#pragma xmp template t[20][20]
#pragma xmp distribute t[block][block] onto p
int a[20][20];
#pragma xmp align a[i][j] with t[i][j]

#pragma xmp loop (i,j) on t[i][j]
for(int i=0;i<20;i++){
  for(int j=0;j<20;j++){
      a[i][j] = func(i, j);
  }
}
```

[F]

```
!$xmp nodes p(2,4)
!$xmp template t(20,20)
!$xmp distribute t(block,block) onto p
integer :: a(20,20);
!$xmp align a(j,i) with t(j,i)

!$xmp loop (i,j) on t(j, i)
do i=1, 20
  do j=1, 20
    a(j,i) = func(j,i)
  end do
end do
```

Loop directive with reduction clause

- The reduction clause
  - reduces the value on each node with the specified operation when ending the loop.
  - Operations: +, *, -, &, |, ^, &&, ||, max, min, firstmax, firstmin, lastmax, lastmin

[C]

```
#pragma xmp loop on t[i] reduction(+:s)
for(int i=0;i<20;i++){
    s = s + i;
}
```

[F]

```
!$xmp loop on t(i) reduction(+:s)
do i=1, 20
  s = s + i
end do
```

The variables s on all nodes are summed up and updated to the value when ending the loop-statement.

Collaboration with OpenMP

[C]

```
#pragma xmp loop on t[i]
#pragma omp parallel for
for(int i=0;i<20;i++){
    a[i] = i;
}
```

[F]

```
!$xmp loop on t(i)
!$omp parallel do
do i=1, 20
   a(i) = i
end do
!$omp end parallel do
```

[C]

```
#pragma omp parallel for
#pragma xmp loop on t[i]
for(int i=0;i<20;i++){
    a[i] = i;
}
```

[F]

```
!$omp parallel do
!$xmp loop on t(i)
do i=1, 20
   a(i) = i
end do
!$omp end parallel do
```

- The order of the XMP loop directive and the OpenMP directive does not matter.

Task directive

- Assigns the following code block to the specified node(s).

[C]

```
#pragma xmp nodes p[*]
 :
#pragma xmp task on p[0]
{
   func_a();
}
#pragma xmp task on p[1]
{
   func_b();
}
```

Node p[0] executes func_a.

Node p[1] executes func_b.

[F]

```
!$xmp nodes p(*)
 :
!$xmp task on p(1)
   call func_a()
!$xmp end task

!$xmp task on p(2)
   call func_b()
!$xmp end task
```

Node p(1) executes func_a.

Node p(2) executes func_b.

Task directive

- Assigns the following code block to the specified node(s).

[C]

```
#pragma xmp nodes p[100]
  :
#pragma xmp task on p[0:50]
{
   func_a();
}
```

p[0] to p[49] execute func_a.

node-name[*base* : *length*]

[F]

```
!$xmp nodes p(100)

!$xmp task on p(1:50)
   call func_a()
!$xmp end task
```

p(1) to p(50) execute func_a().

node-name(base : *end)*

Agenda in the morning session

- Overview
- XcalableMP language
  - Global-view
    - Declare distributed array
    - Parallelize loop statement
    - Perform communication
  - Local-view
- Omni XcalableMP compiler
- How to install Omni XcalableMP compiler (Hands-on)
  - Create Hello World program

bcast and reduction directives

- bcast directive
  - broadcasts the specified data among nodes

[C]

```
#pragma xmp bcast (b)
#pragma xmp bcast (b) from p[2]
```

[F]

```
!$xmp bcast (b)
!$xmp bcast (b) from p(3)
```

- reduction directive
  - Performs a reduction operation (+, *, max, min, …) among nodes.

[C]

```
#pragma xmp reduction (+:b)
```

[F]

```
!$xmp reduction (+:b)
```

If “from” clause is omitted, the directive broadcast a variable located in root node (p[0] in C or p(1) in Fortran).

barrier directive

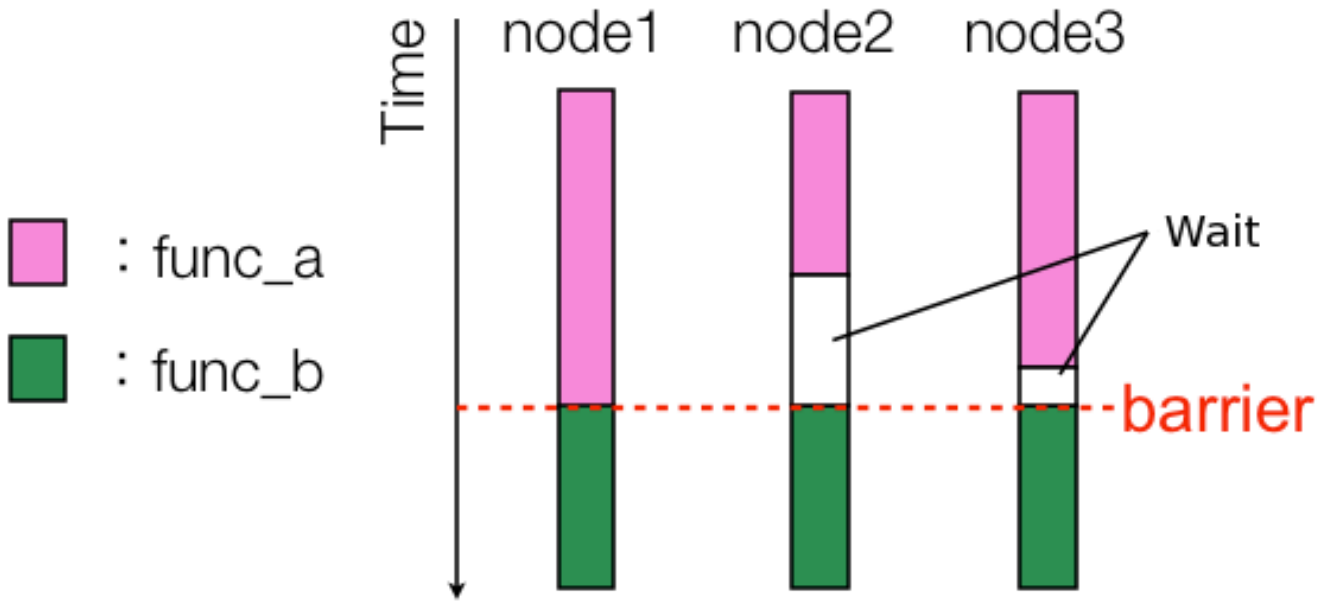
- barrier directive
  - barrier operation is performed

C

```
func_a();
#pragma xmp barrier
func_b();
```

Fortran

```
call func_a()
!$xmp barrier
call func_b()
```

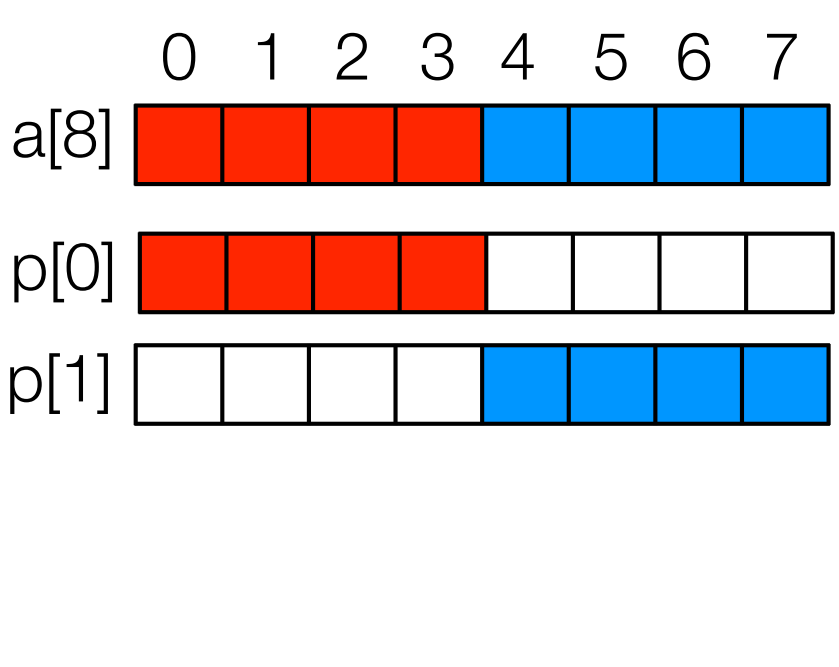


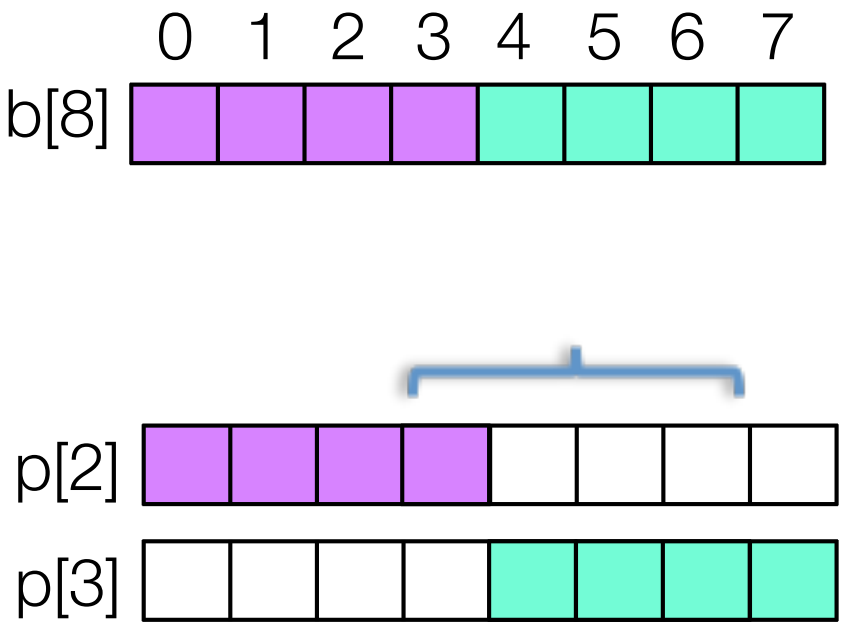
gmove directive

- Communication for distributed array
  - Programmer doesn't need to know where each data is distributed

[C]
```
#pragma xmp gmove
a[2:4] = b[3:4];
```

array-name[*base* : *length*];





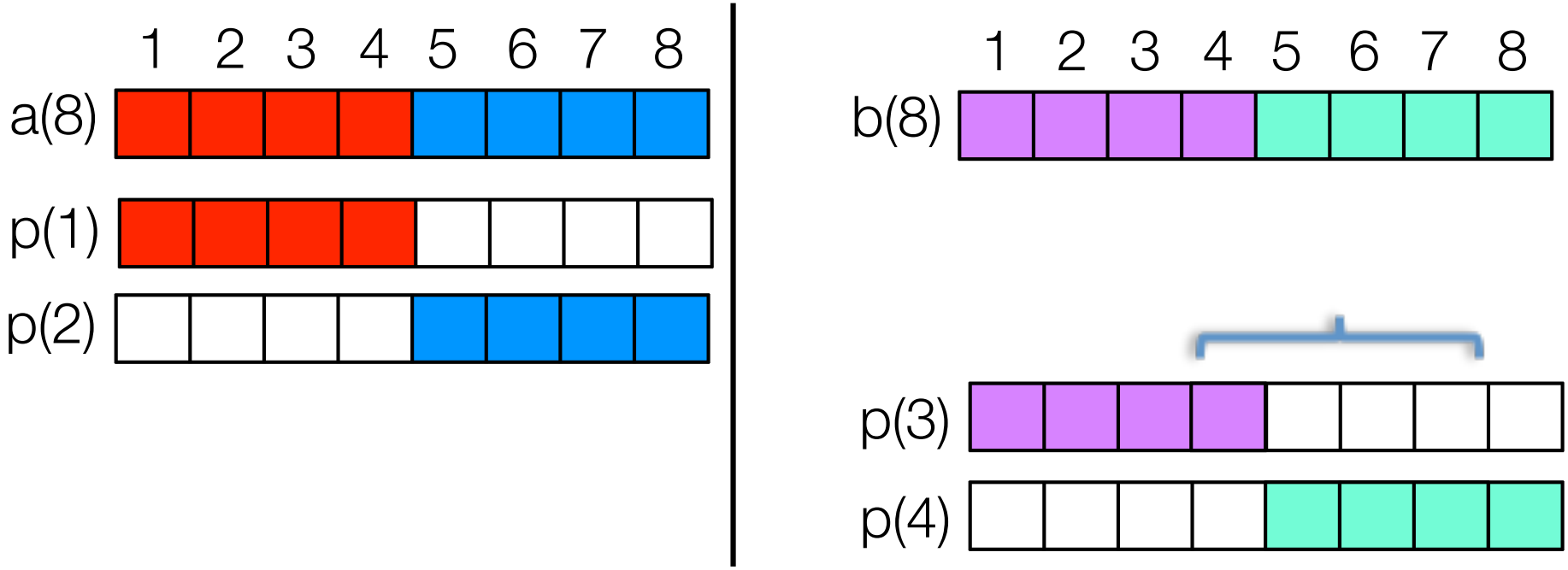
gmove directive

- Communication for distributed array
  - Programmer doesn't need to know where each data is distributed

```
!$xmp gmove
a(3:6) = b(4:7)
```

[F]

array-name(base : end *)*



Shadow and reflect directive

- These directives are used to develop stencil applications
- Shadow directive adds shadow area to distributed array
- Reflect directive updates the shadow area

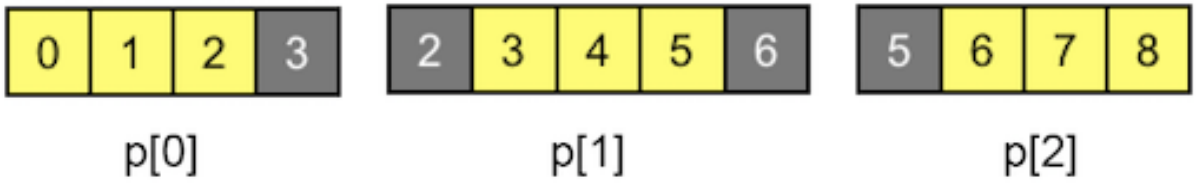
[C]

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

[F]

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

Add shadow areas of size one at both the lower and upper bounds of a[].



The shadow directive creates a shadow area (gray cell) at the upper and lower bounds of array a[].

Shadow and reflect directive

- These directives are used to develop stencil applications
- Shadow directive adds shadow area to distributed array
- Reflect directive updates the shadow area

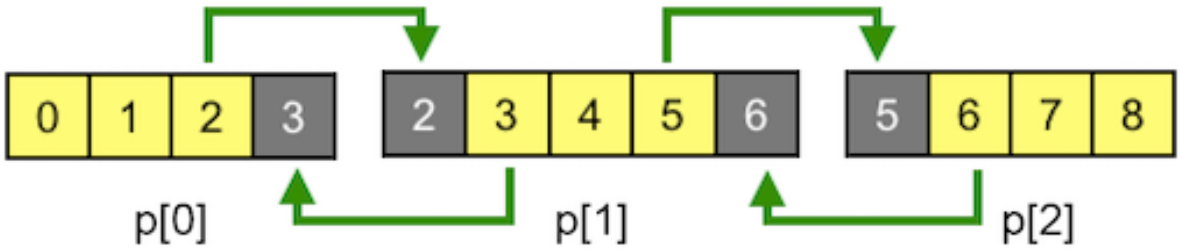
[C]

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

[F]

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

Add shadow areas of size one at both the lower and upper bounds of a[].



The reflect directive synchronizes the shadow area. The directive generates communication between adjacent nodes.

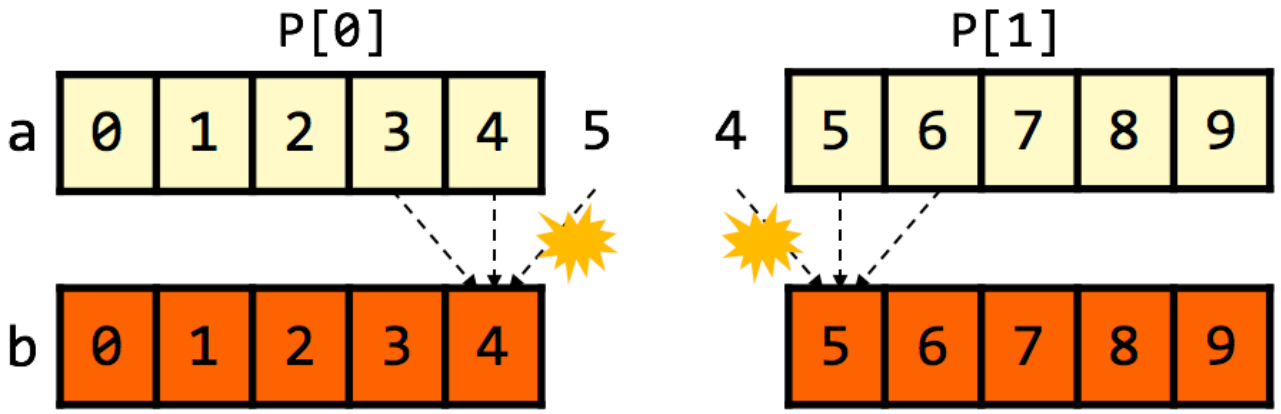
Example of shadow/reflect

[C]

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

[F]

```
!$xmp loop on t(i)
do i = 2, 8
  b(i) = a(i-1) + a(i) + a(i+1)
end do
```



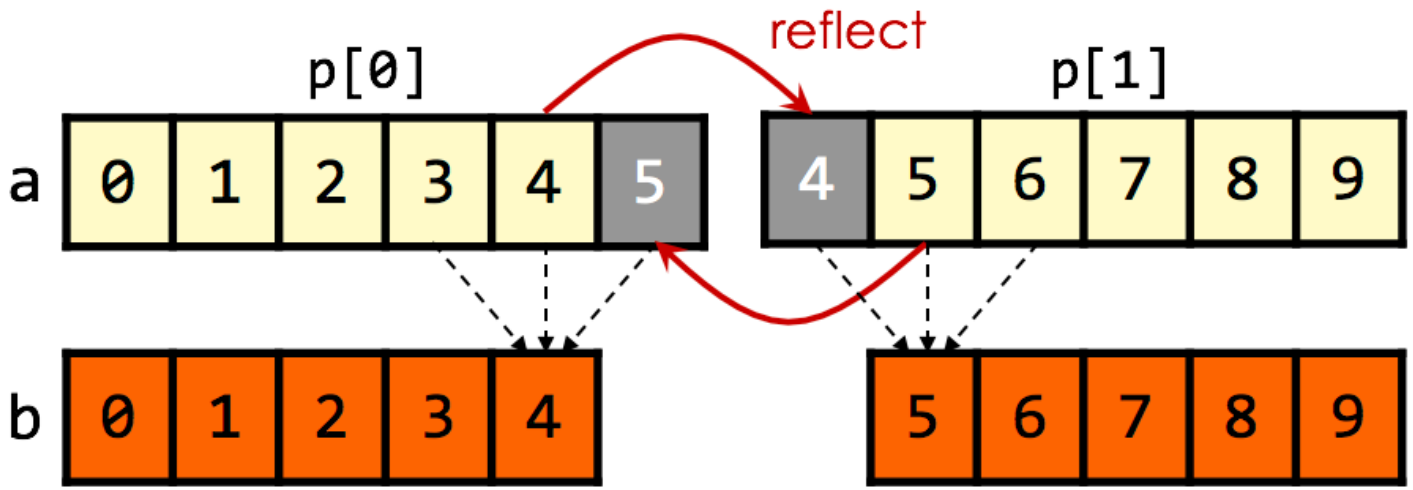
Example of shadow/reflect

[C]

```
#pragma xmp shadow a[1:1]
...
#pragma xmp reflect (a)
#pragma xmp loop on t[i]
for(int i=1;i<9;i++){
  b[i] = a[i-1] + a[i] + a[i+1];
}
```

[F]

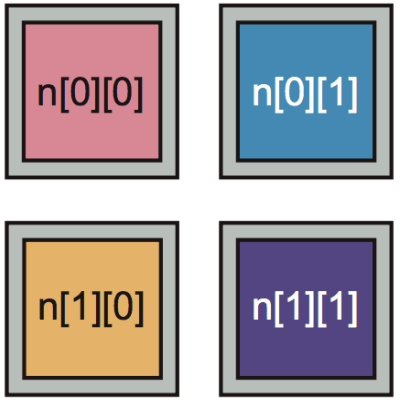
```
!$xmp shadow a(1:1)
...
!$xmp reflect (a)
!$xmp loop on t(i)
do i = 2, 8
  b(i) = a(i-1) + a(i) + a(i+1)
end do
```



Example of shadow/reflect

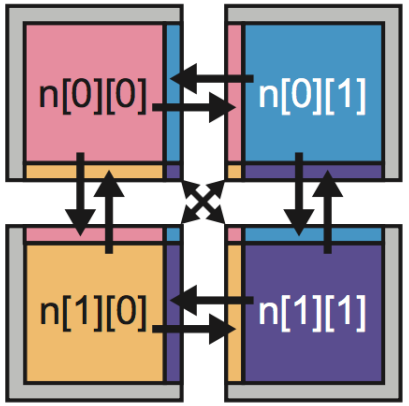
[C]
```
#pragma xmp shadow a[1:1][1:1]
```

[F]
```
!$xmp shadow a(1:1,1:1)
```



[C]
```
#pragma xmp reflect (a)
```

[F]
```
!$xmp reflect (a)
```



Agenda in the morning session

- Overview
- XcalableMP language
  - Global-view
    - Declare distributed array
    - Parallelize loop statement
    - Perform communication
  - Local-view
- Omni XcalableMP compiler
- How to install Omni XcalableMP compiler (Hands-on)
  - Create Hello World program

Coarray in XMP/Fortran

- XMP includes the coarray feature imported from Fortran 2008 for the local-view programming.
- Basic idea: data declared as a coarray can be accessed by remote nodes.
- Coarray in XMP/Fortran is fully compatible with Fortran 2008.

[F]

```
real a(8)
real b(8)[*]

if(this_image() == 1) then
  b(6)[3] = b(2)
  a(4) = b(3)[2]
end if

sync all
```

b() is declared as a coarray

image 1 puts b(2) to b(6) at node 3

image 1 gets b(3) from node 2 to a(4)

Synchronization

Coarray in XMP/C

- Coarray can be used in XMP/C
  - Declaration
  - Put
  - Get
  - Synchronization

```
double b[8]:[*];
```

```
b[6]:[3] = b[2];
```

```
a[4] = b[3]:[2];
```

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

[C]

```
double a[8];
double b[8]:[*];

if(xmpc_this_image() == 1){
  b[6]:[3] = b[2];
  a[4] = b[3]:[2];
}

xmpc_sync_all(NULL);
```

[F]

```
real a(8)
real b(8)[*]

if(this_image() == 1) then
  b(6)[3] = b(2)
  a(4) = b(3)[2]
end if

sync all
```

Subarray in XMP/C

- To put/get multiple elements, XMP/C provides the subarray
- The syntax is the same as that in Intel Cilk and OpenACC

array-name[*base* : *length* : *step*]

[C]

```
if(xmpc_this_image() == 1){
  a[10:5]:[3] = b[0:5];
  a[10:5:2]:[3] = b[0:5:2];
  a[:]:[3] = b[:];
}
```

b[0]-b[4] elements are put to a[10]-a[14] elements at image 3

b[0], b[2], b[4], b[6], and b[8] elements are put to a[10], a[12], a[14], a[16], and a[18] elements at image 3

All elements of b[] are put to all elements of a[] at image 3

Subarray in XMP/Fortran

- The subarray is the same as normal subarray in Fortran

array-name[*base* : *last* : *step*]

[F]

```
if(this_image() == 1) then
  a(10:14)[3] = b(1:5)
  a(10:18:2)[3] = b(1:9:2)
  a(:)[3] = b(:)
end if
```

b(1)-b(5) elements are put to a(10)-a(14) elements at image 3

b(1), b(3), b(5), b(7), and b(9) elements are put to a(10), a(12), a(14), a(16), and a(18) elements at image 3

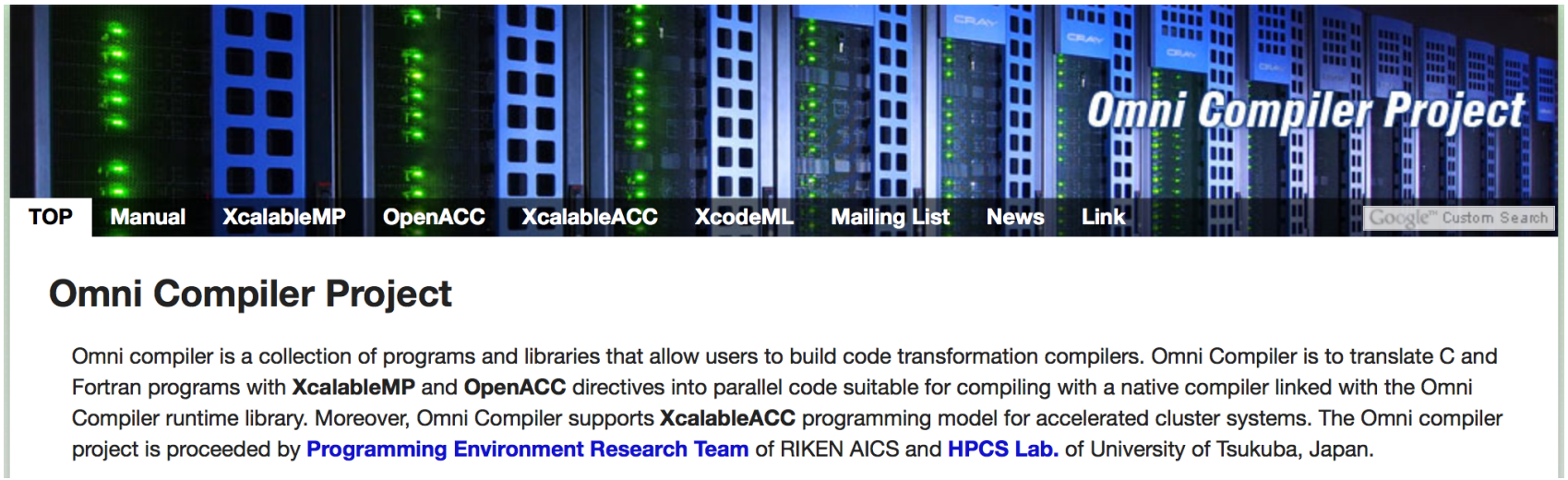
All elements of b() are put to all elements bf a() at image 3

Agenda in the morning session

- Overview
- XcalableMP language
  - Global-view
    - Declare distributed array
    - Parallelize loop statement
    - Perform communication
  - Local-view
- Omni XcalableMP compiler
- How to install Omni XcalableMP compiler (Hands-on)
  - Create Hello World program

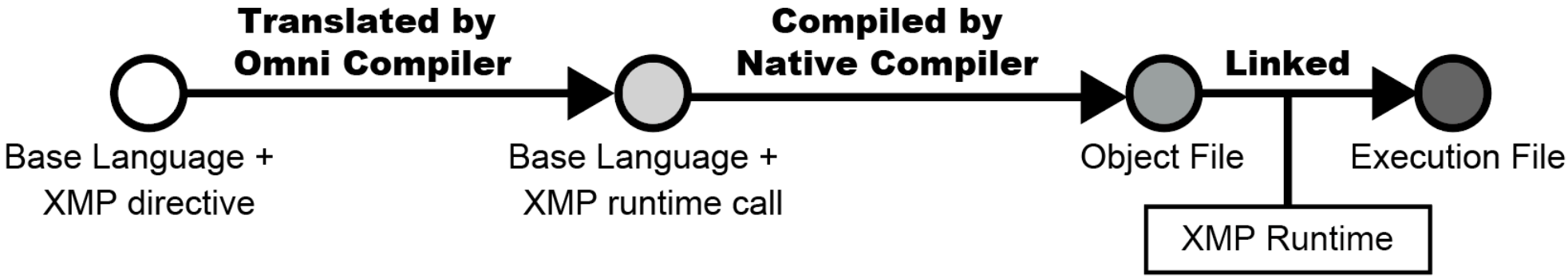
Omni Compiler

- Support XMP, OpenACC, XcalableACC
- Developed by RIKEN AICS and University of Tsukuba, Japan
- Open Source Software on GitHub
- Source-to-source Compiler
- The latest version 1.2.1 is available at http://omni-compier.org



Omni Compiler

- Omni XMP compiler = Translator + Runtime
  - In the runtime, global-view functions are implemented in MPI
  - In the runtime, local-view functions are implemented in MPI, GASNet, or FJRDMA
    - User selects one of them in installation



Agenda in the morning session

- Overview
- XcalableMP language
  - Global-view
    - Declare distributed array
    - Parallelize loop statement
    - Perform communication
  - Local-view
- Omni XcalableMP compiler
- How to install Omni XcalableMP compiler (Hands-on)
  - Create Hello World program

Install Omni Compiler

- Please visit http://omni-compiler.org
- Download the latest version omni-compiler-1.2.1
- Expand the archive on the cluster
  - $ tar xvfj omnicompiler-1.2.1.tar.bz2
- Install
  - $ module load intelmpi/5.0.1
  - $ cd omnicompiler-1.2.1
  - $ ./configure --prefix=(your install path)
  - $ make
  - $ make install
  - $ export PATH=(your install path)/bin:$PATH

Hello World

[C]

```
$ emacs hello.c
```

```
#include <stdio.h>
#include <xmp.h>
#pragma xmp nodes p[*]
int main(){
  printf("Hello World on node %d\n",
  xmpc_node_num());
  return 0;
}
```

```
$ xmpcc hello.c -o hello
$ mpirun -np 2 ./hello
```

[F]

```
$ emacs hello.f90
```

```
program hello
!$xmp nodes p(*)
write(*,*) "Hello World on node ",
xmp_node_num()
end program
```

```
$ xmpf90 hello.f90 -o hello
$ mpirun -np 2 ./hello
```