



Highly productive  
parallel programming language

*XcalableMP*

---

# Outline

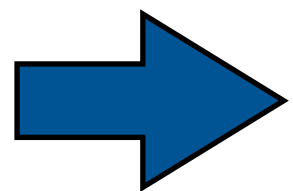
---

- Background
- About XcalableMP Programming
- XcalableMP Directive
  - Data Mapping
  - Work Mapping
  - Communication/Synchronization

# Background

---

- MPI is widely used as a parallel programming model on distributed memory systems
- However, writing MPI programs is often a time-consuming and a complicated process
- Another programming model is needed !!
  - High performance
  - Easy to program



Development of XcalableMP

# What's XcalableMP?

---

- XcalableMP(XMP) is a new programming model and language for distributed memory systems
- XMP is proposed by XcalableMP Specification Working Group(XMP WG)
  - XMP WG is a special interest group, which is organized to make a draft on “petascale” parallel language
  - XMP WG consists of members from academia (U. Tsukuba, U. Tokyo, Kyoto U. and Kyusyu U.), research labs(RIKEN, NIFS, JAXA, JAMSTEC/ES) and industries(Fujitsu, NEC, Hitachi)
- XMP Specification version 1.1 released !!
  - <http://www.xcalablemp.org/>



**Directive-based language eXtension for  
Scalable and performance-aware Parallel Programming**

[日本語](#)

[\[home\]](#) [\[publications\]](#) [\[download\]](#) [\[projects\]](#)

## What's XcalableMP

Although MPI is the de facto standard for parallel programming on distributed memory systems, writing MPI programs is often a time-consuming and complicated process. XcalableMP is a directive-based language extension which allows users to develop parallel programs for distributed memory systems easily and to tune the performance by having minimal and simple notations.

The specification is being designed by XcalableMP Specification Working Group which consists of members from academia and research labs to industries in Japan.

The features of XcalableMP are summarized as follows:

- XcalableMP supports typical parallelization based on the data parallel paradigm and work mapping under "global view programming model", and enables parallelizing the original sequential code using minimal modification with simple directives, like OpenMP. Many ideas on "global-view" programming are inherited from HPF (High Performance Fortran).
- The important design principle of XcalableMP is "performance-awareness". All actions of communication and synchronization are taken by directives, different from automatic parallelizing compilers. The user should be aware of what happens by XcalableMP directives in the execution model on the distributed memory architecture.
- XcalableMP also includes a CAF-like PGAS (Partitioned Global Address Space) feature as "local view" programming.
- Extension of existing base languages with directives is useful to reduce rewriting and educational costs. XcalableMP APIs are defined on C and Fortran 95 as a base language.
- For flexibility and extensibility, the execution model allows to combine with explicit MPI coding for more complicated and tuned parallel codes and libraries.
- For multi-core and SMP clusters, OpenMP directives can be combined into XcalableMP for thread programming inside each node as a hybrid programming model.(Under discussion)

XcalableMP is being designed based on the experiences of HPF, Fujitsu XPF (VPP Fortran) and OpenMPD.

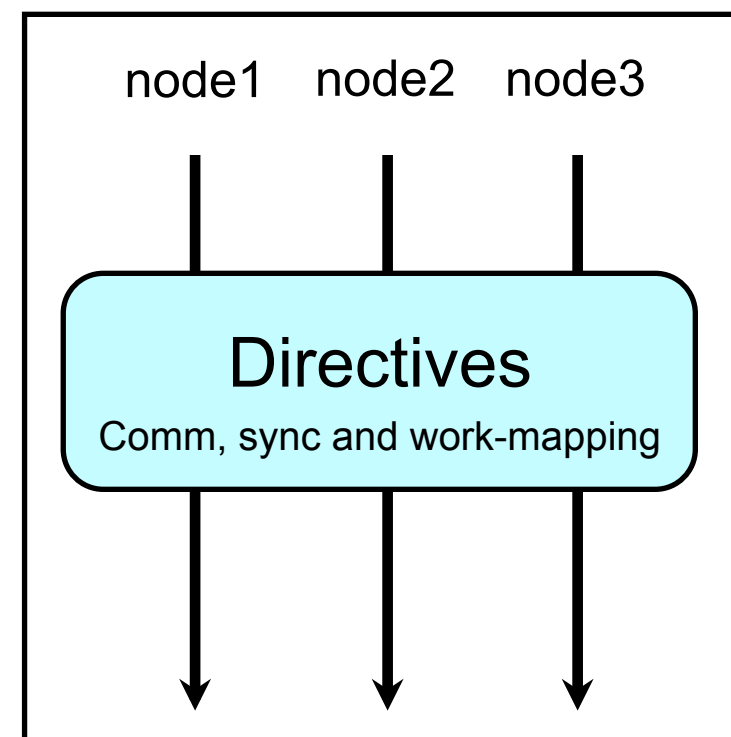
<http://www.xcalablemp.org>



# Overview of XMP

---

- XMP is a directive-based language extension like OpenMP and HPF based on C and Fortran95
  - To reduce code-writing and educational costs
- “Scalable” for Distributed Memory Programming
- A thread starts execution in each node independently (as in MPI)
- “Performance-aware” for explicit communication, sync. and work-sharing
  - All actions occur when directives are encountered
  - All actions are taken by directives for being “easy-to-understand” in performance tuning (different from HPF)



# XMP Code Example

```
int array[100];
#pragma xmp nodes p(*)
#pragma xmp template t(0:99)
#pragma xmp distribute t(block) onto p
#pragma xmp align array[i] with t(i)
main(){
#pragma xmp loop on t(i) reduction(+:res)
    for(i = 0; i < 100; i++){
        array[i] = func(i);
        res += array[i];
    }
}
```

**data  
mapping**

**work  
mapping &  
reduction**

Programmer adds **XMP directives** to serial code  
<serial incremental parallelization>

# The same code written in MPI

```
int array[100]; main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    dx = 100/size;
    llimit = rank * dx;
    if(rank != (size -1)) ulimit = llimit + dx;
    else ulimit = 100;
    tmp = 0;
    for(i=llimit; i < ulimit; i++){
        array[i] = func(i);
        tmp += array[i];
    }
    MPI_Allreduce(&tmp, &res, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Finalize();
}
```



# Programming Model

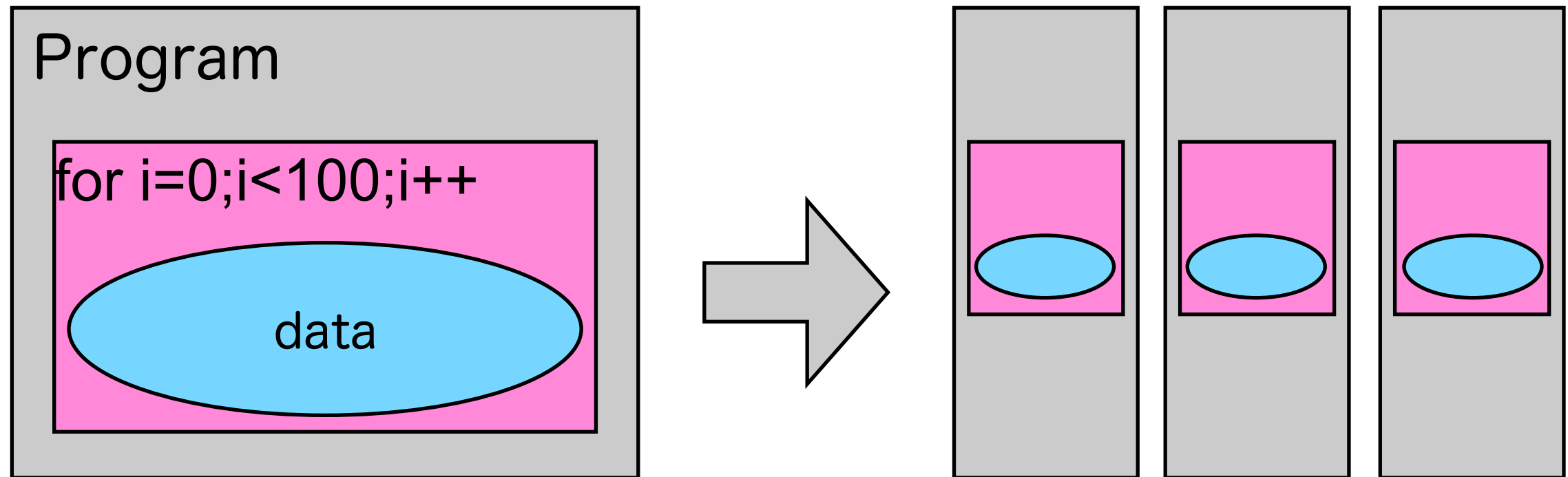
---

- Global View Model (like as HPF)
  - programmer describes data distribution, work mapping and inter-node comm. with adding directives to serial code
  - support typical techniques for data-mapping and work-mapping
  - rich communication and sync. directives, such as “shadow”, “reflect” and “gmove”
- Local View Model (like as Coarray Fortran)
  - enable programmer to communicate with node index
  - one-sided communication using language extension (also defined C)

# Data Mapping(1/3)

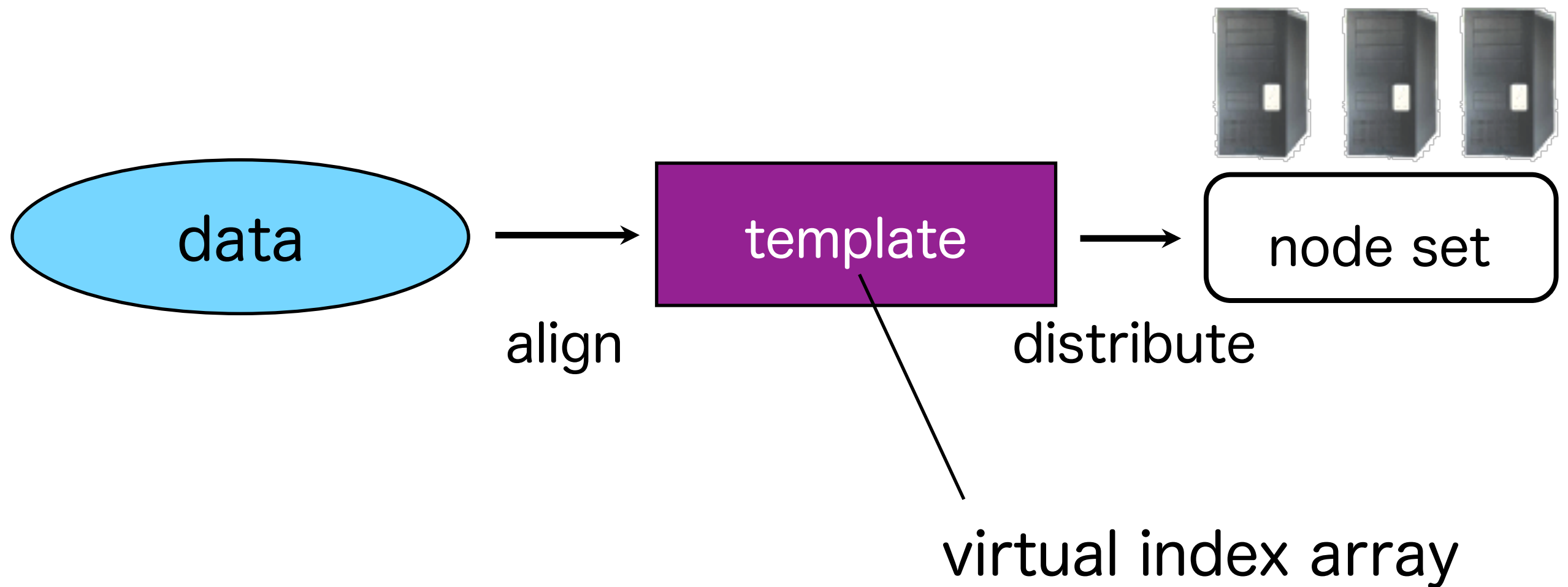
---

- To distribute data onto each node
- It is important not only work mapping, but also distribution of data



# Data Mapping(2/3)

---



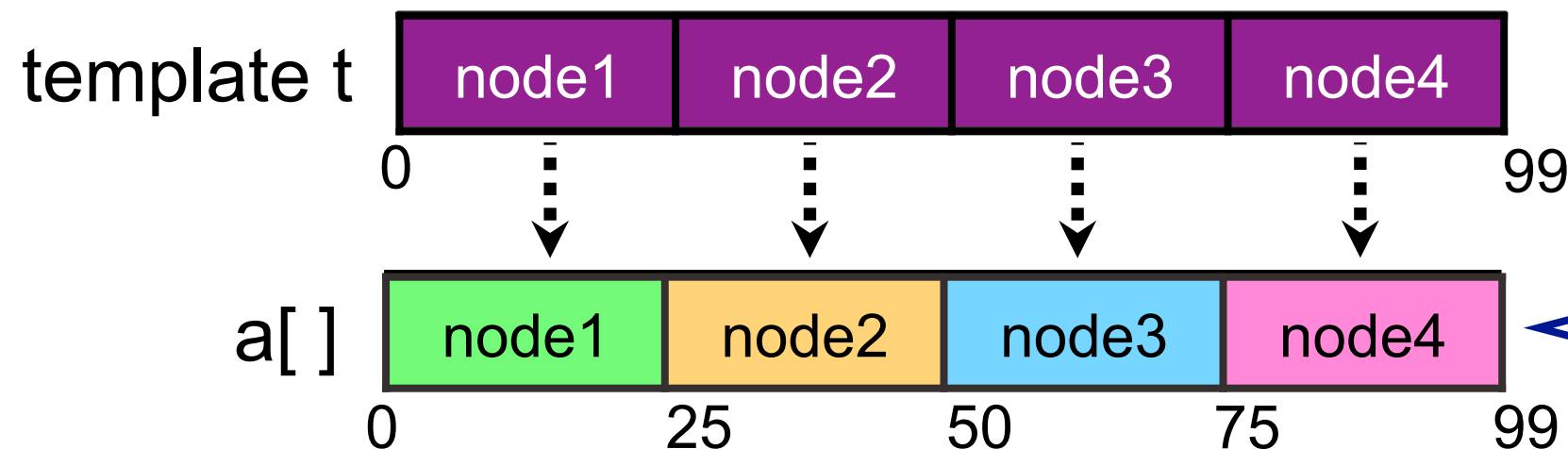
# Data Mapping(3/3)

```
int a[100];  
#pragma xmp template t(0:99)  
#pragma xmp nodes p(4)  
#pragma xmp distribute t(block) onto p  
#pragma xmp align a[i] with t(i)
```

The directives specify  
a data distribution  
among nodes

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

Execute “for” loop in parallel with  
affinity to array distribution



# Data Mapping(summary)

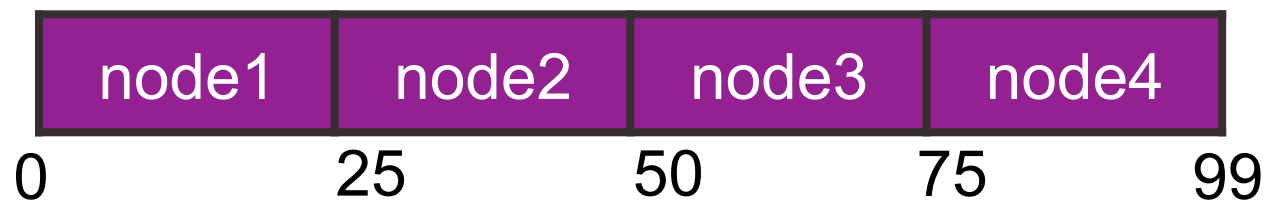
## 1. Create template

`#pragma xmp template t(0:99)`



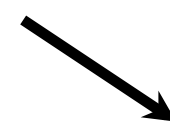
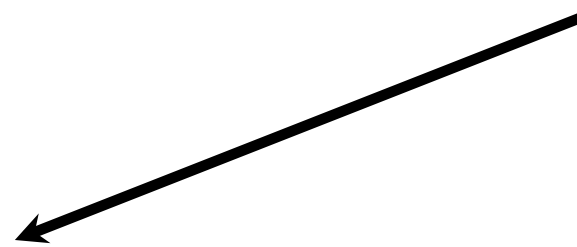
## 3. Distribute template onto node set

`#pragma xmp distribute t(block) onto p`



## 2. Define Node set

`#pragma xmp nodes p(4)`



## 4. Align data with template

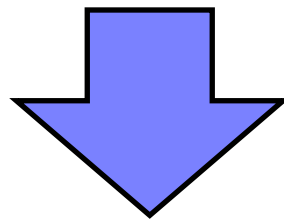
`int a[100];`

`#pragma xmp align a[i] with t(i)`

# Example Data Mapping

## Example 1

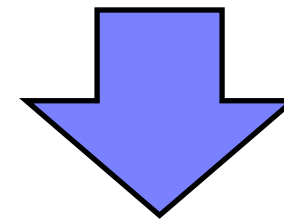
```
#pragma xmp template t(0:19)
#pragma xmp nodes p(4)
#pragma xmp distribute t(block) onto p
```



node	index
p(1)	0, 1, 2, 3, 4
p(2)	5, 6, 7, 8, 9
p(3)	10, 11, 12, 13, 14
p(4)	15, 16, 17, 18, 19

## Example 2

```
#pragma xmp template t(0:19)
#pragma xmp nodes p(4)
#pragma xmp distribute t(cyclic) onto p
```



node	index
p(1)	0, 4, 8, 12, 16
p(2)	1, 5, 9, 13, 17
p(3)	2, 6, 10, 14, 18
p(4)	3, 7, 11, 15, 19

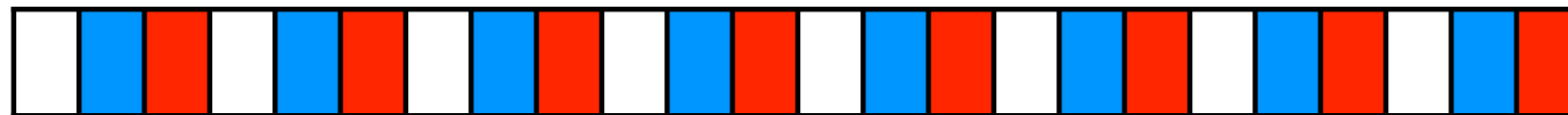


# Example Data Mapping

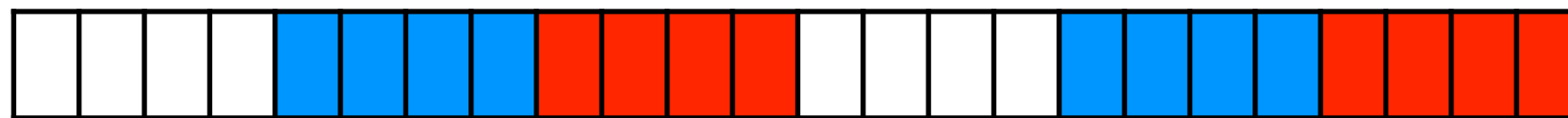
---



**block**



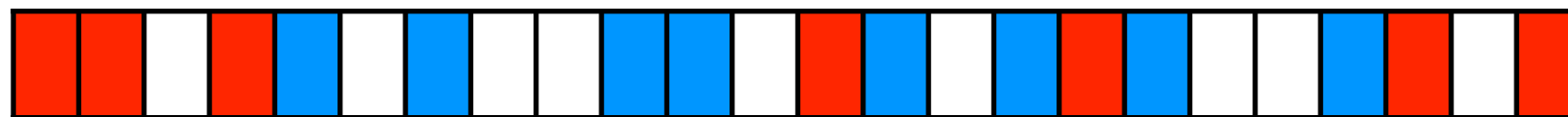
**cyclic**



**block-cyclic**  
(block size = 4)



**generalized-block**



**indirect**



# Loop directive(1/2)

---

```
#pragma xmp template t(0:19)
#pragma xmp nodes p(4)
#pragma xmp distribute t(block) onto p
int a[20];
#pragma xmp align a[i] with t(i)

int main(void){
    int i;
    #pragma xmp loop on t(i)
    for(i=0;i<20;i++)
        a[ i ] = func( i );

    return(0);
}
```

Loop directive is  
inserted before loop  
statement




Each node computes  
in parallel

# Loop directive(2/2)

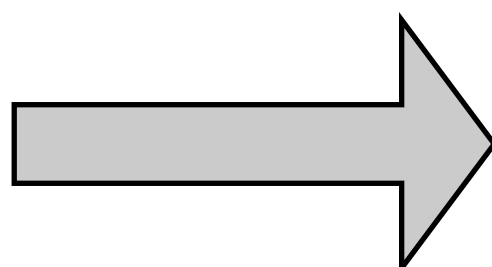
```
int main(void){  
    int i, sum = 0;  
  
    #pragma xmp loop on t(i) reduction(+:sum)  
    for(i=0;i<20;i++)  
        sum += i;  
  
    return(0);  
}
```

reduction operation  
+, \*, MAX, MIN, and  
so on



Node	Calculation	value of <i>sum</i>
p(1)	0+1+2+3+4	10
p(2)	5+6+7+8+9	35
p(3)	10+11+12+13+14	60
p(4)	15+16+17+18+19	85

reduction



190

# Task Directive

`#pragma xmp task on [nodes-ref | template-ref]`

- The processing immediately after this directive executes the node specified by *nodes-ref* or *template-ref*

```
#pragma xmp nodes p(2)
```

```
#pragma xmp tasks
```

```
{
```

```
#pragma xmp task on p(1)
```

```
    func_a();
```

```
#pragma xmp task on p(2)
```

```
    func_b();
```

```
}
```

p(1) executes func\_a() and  
p(2) executes func\_b() one  
at a time

p(1)

func\_a();

p(2)

func\_b();

time



# Other directives

---

Directive	Function
reduction	Aggregation
bcast	Broadcast
barrier	Synchronization
shadow/reflect	Create shadow region/sync.
gmove	Transfer for distributed data
coarray	One-sided comm. (Put/Get)

# reduction Directive

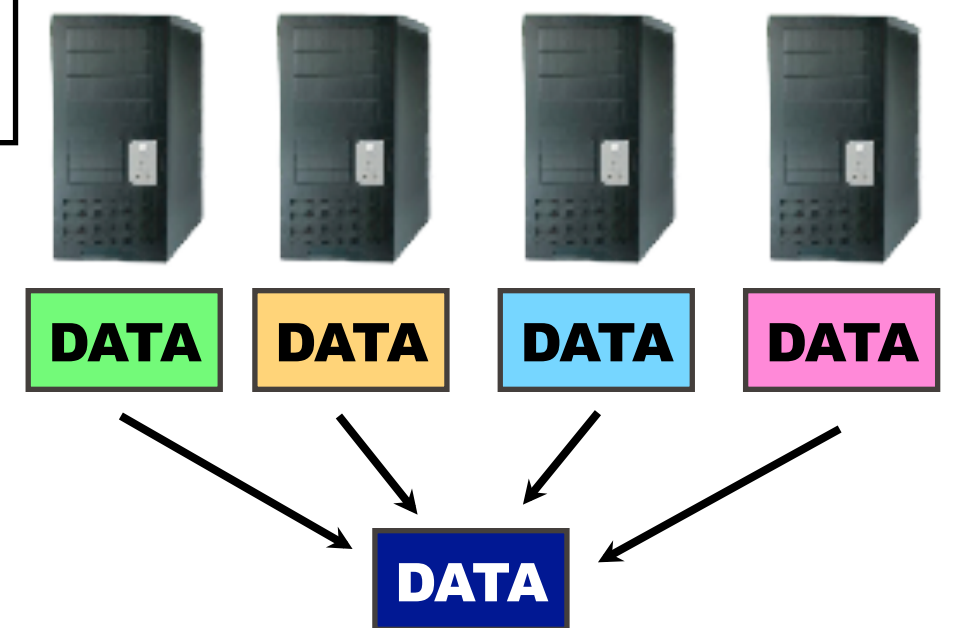
```
#pragma xmp reduction(reduction-kind : variable[, variable]...)  
[on nodes-ref / template-ref]
```

- This directive performs data aggregation operation

```
sum=0;  
#pragma xmp loop on t(i) reduction(+:sum)  
  for(i=0;i<20;i++)  
    sum += func( i );
```

```
sum = 0;  
#pragma xmp loop on t(i)  
  for(i=0;i<20;i++)  
    sum += func( i );
```

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



Max, Sum and so on



# bcast directive

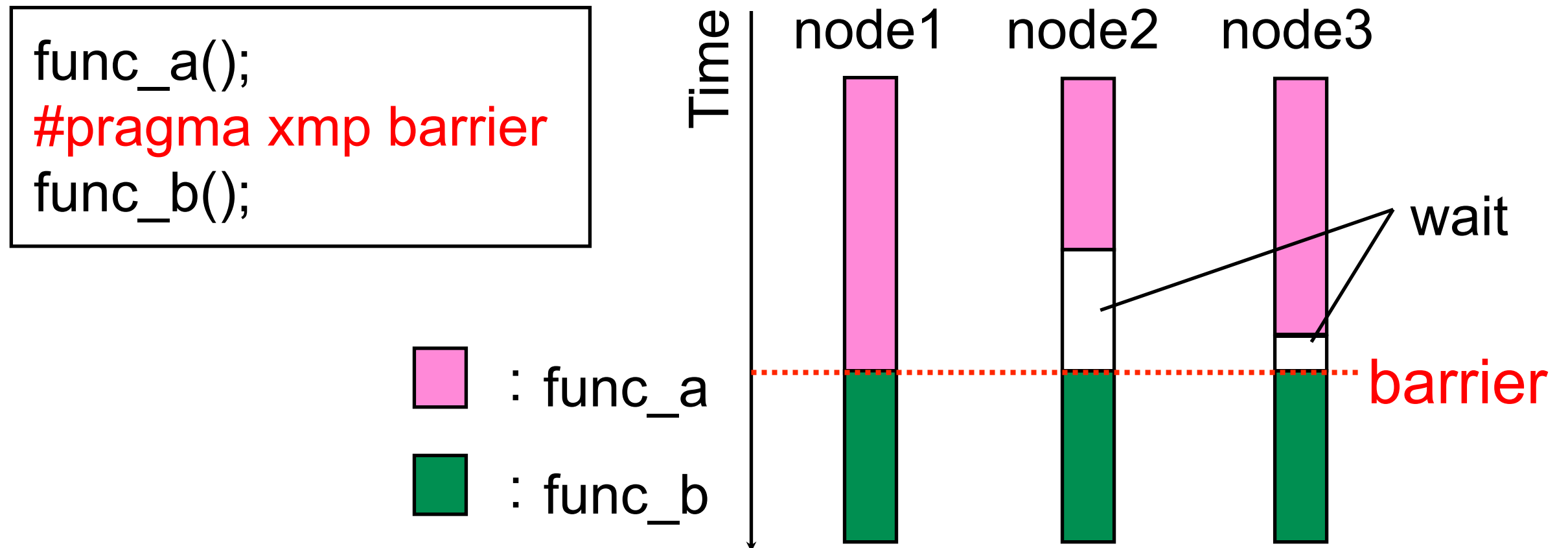
```
#pragma xmp bcast variable [, variable] ...  
[from nodes-ref] [on nodes-ref | template-ref]
```

- This directive broadcasts data
- It specifies the transfer source with the *from* clause.  
When omitted, p(1) is assigned
- It specifies the transfer destination with *the* on clause.  
When omitted, the transfer is to all nodes

```
#pragma xmp nodes p(4)  
...  
#pragma xmp bcast (a) from p(1)
```

# barrier directive

```
#pragma xmp barrier [on nodes-ref | template-ref]
```

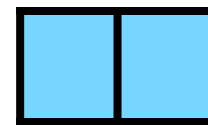
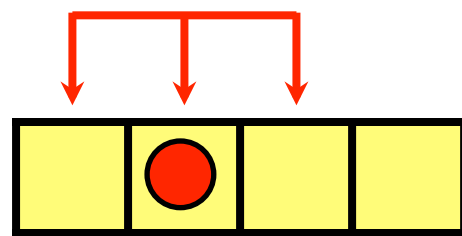


In this case, all nodes wait until func\_a() ends.

# shadow directive

```
#pragma xmp shadow array-name[shadow-width  
[, shadow-width] ...]
```

- Define the area used to temporarily store the values of neighboring elements
- *shadow-width* is the size of the reference area

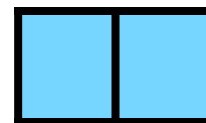
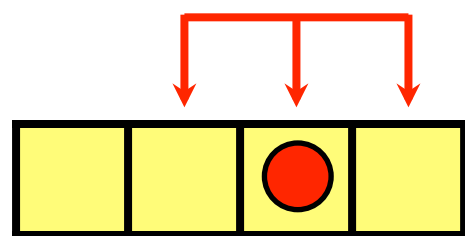


Neighboring elements is needed  
to calculate an own element!!

# shadow directive

```
#pragma xmp shadow array-name[shadow-width  
[, shadow-width] ...]
```

- Define the area used to temporarily store the values of neighboring elements
- *shadow-width* is the size of the reference area

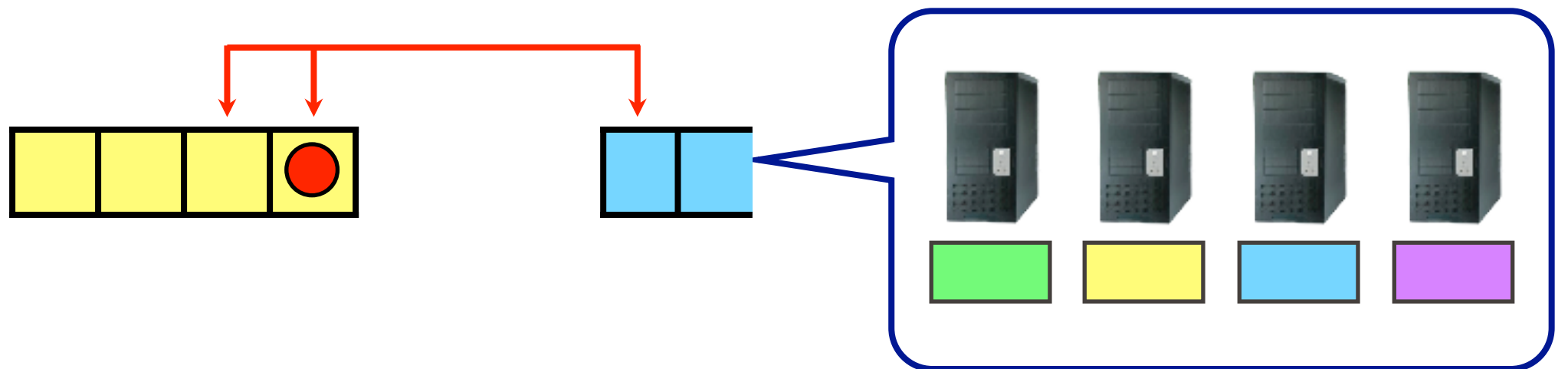


Neighboring elements is needed  
to calculate an own element!!

# shadow directive

```
#pragma xmp shadow array-name[shadow-width  
[, shadow-width] ...]
```

- Define the area used to temporarily store the values of neighboring elements
- *shadow-width* is the size of the reference area

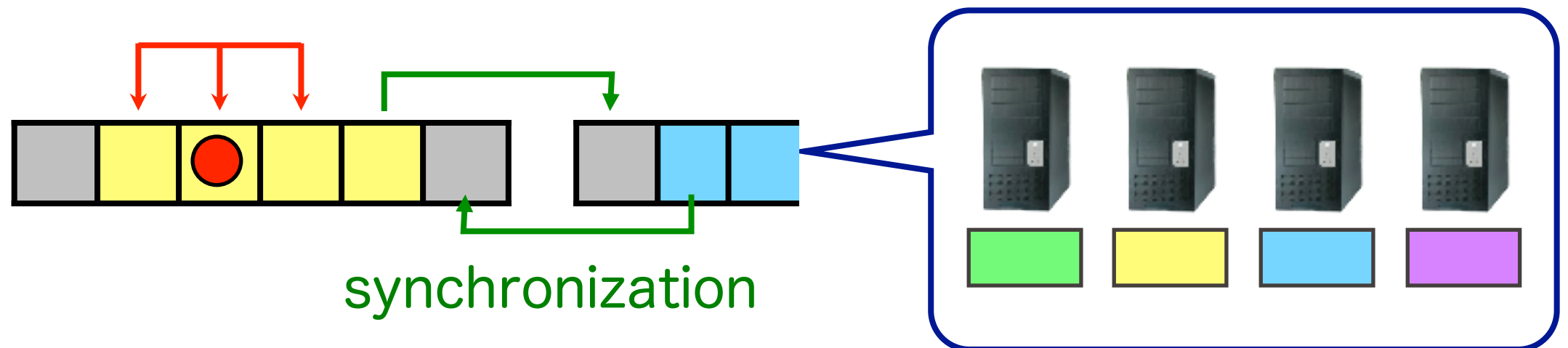


Neighboring elements is needed  
to calculate an own element!!

# shadow directive

```
#pragma xmp shadow array-name[shadow-width  
[, shadow-width] ...]
```

- Define the area used to temporarily store the values of neighboring elements
- *shadow-width* is the size of the reference area



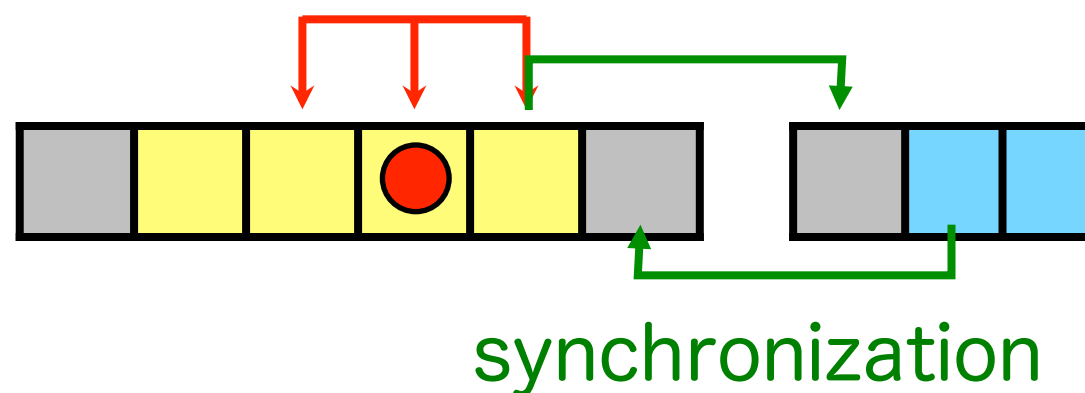
Neighboring elements is needed  
to calculate an own element!!



# shadow directive

```
#pragma xmp shadow array-name[shadow-width  
[, shadow-width] ...]
```

- Define the area used to temporarily store the values of neighboring elements
- *shadow-width* is the size of the reference area

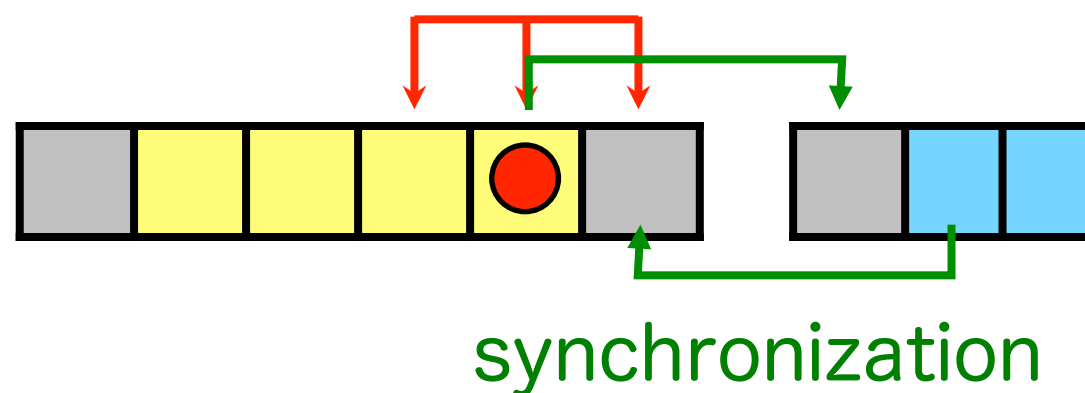


Neighboring elements is needed  
to calculate an own element!!

# shadow directive

```
#pragma xmp shadow array-name[shadow-width  
[, shadow-width] ...]
```

- Define the area used to temporarily store the values of neighboring elements
- *shadow-width* is the size of the reference area



Neighboring elements is needed  
to calculate an own element!!

# Example of shadow directive

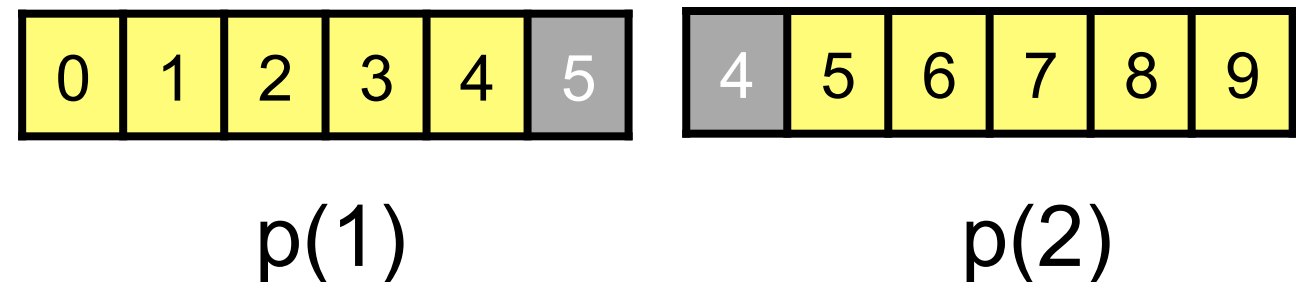
```
#pragma xmp template t(0:9)
#pragma xmp nodes p(2)
int a[10], b[10];
#pragma align [i] with t(i) :: a, b
#pragma xmp shadow a[1:1]
```

```
:
```

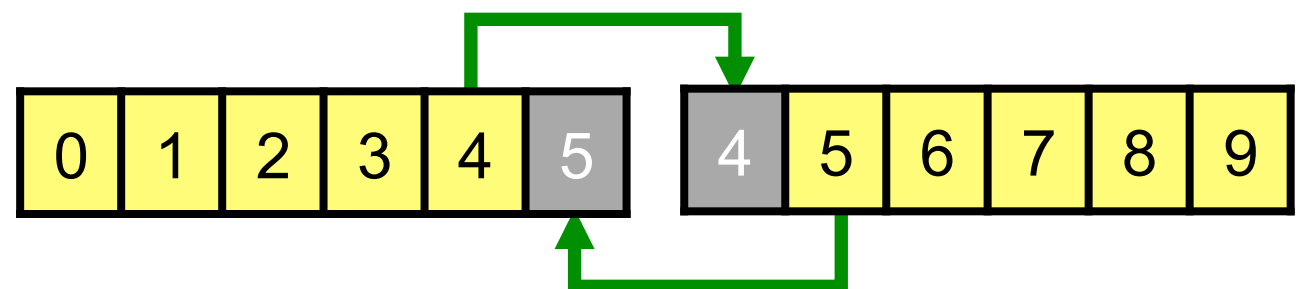
```
#pragma xmp loop on t(i)
for(i=0;i<10;i++)
    a[i] = init(i);    // initialize
#pragma xmp reflect (a)
```

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

The shadow directive creates a sleeve area at the upper and lower bounds of array `a[]`.  
(The gray area is the reference element that is created)



The reflect directive synchronizes the sleeve area. The directive generates communication between adjacent nodes as below.



# gmove directive

```
#pragma xmp gmove [in | out]  
  
<assign statement>
```

- This directive performs data communication with respect to the distributed array
- We extend “array section notation” in C



```
#pragma xmp gmove
```

```
a[0:3] = b[3:3];
```

base length

Programmer doesn't need to know where each data is distributed

# gmove directive

```
#pragma xmp gmove [in | out]  
  
    <assign statement>
```

- This directive performs data communication with respect to the distributed array
- We extend “array section notation” in C



```
#pragma xmp gmove
```

```
a[0:3] = b[3:3];
```

base length

Programmer doesn't need to know where each data is distributed

# coarray function

- We adopt Co-Array as PGAS (Partitioned Global Address Space) feature
- One-sided communication for local data(Put/Get)
- High interoperability with MPI

```
#pragma xmp coarray b
```

```
:
```

```
if(me == 1)
```

```
  a[0:3] = b[3:3]:[2];    // Get
```

indicate node number

node 1



a[10]

b[10]

node 2



a[10]

b[10]

3

6



# coarray function

- We adopt Co-Array as PGAS (Partitioned Global Address Space) feature
- One-sided communication for local data(Put/Get)
- High interoperability with MPI

```
#pragma xmp coarray b
```

```
:
```

```
if(me == 1)
```

```
  a[0:3] = b[3:3]:[2];    // Get
```

indicate node number

node 1



a[10]

b[10]

node 2



a[10]



3

6

# coarray function

- We adopt Co-Array as PGAS (Partitioned Global Address Space) feature
- One-sided communication for local data(Put/Get)
- High interoperability with MPI

```
#pragma xmp coarray b
```

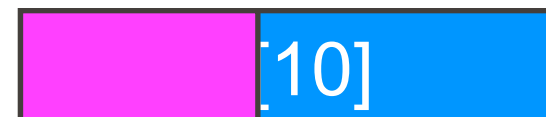
```
:
```

```
if(me == 1)
```

```
  a[0:3] = b[3:3]:[2];    // Get
```

indicate node number

node 1



node 2



3

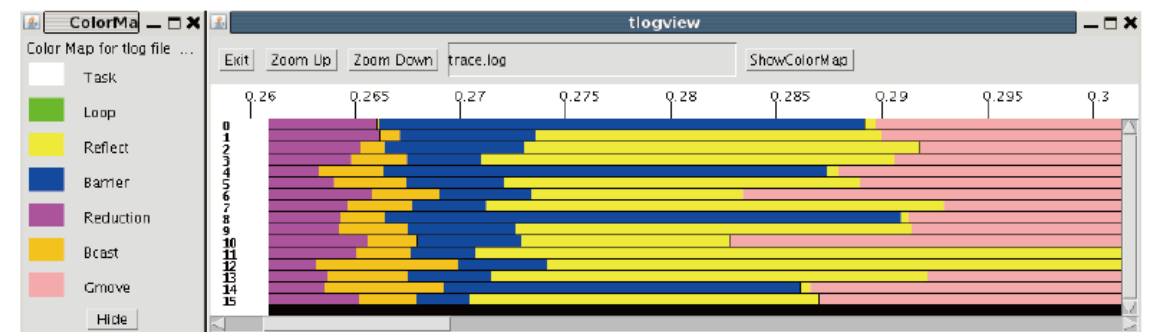
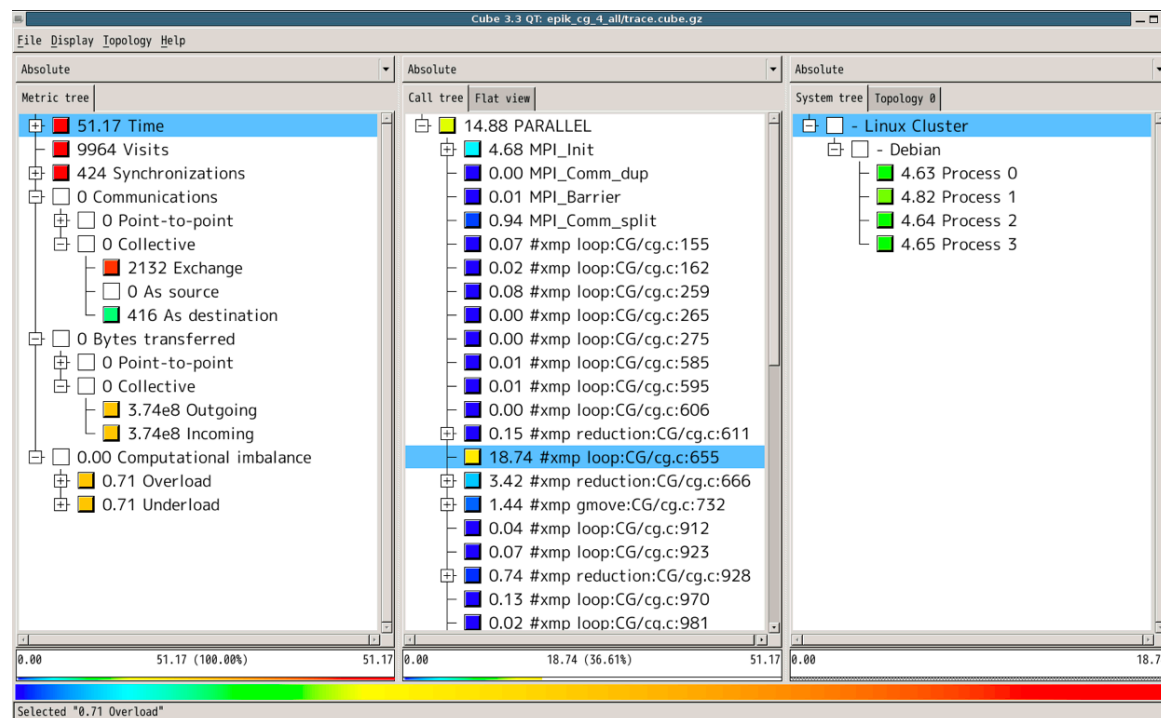
6

# Other Functions

- Easy Hybrid Programming

```
#pragma xmp loop on t(i) threads ←  
for(i=2;i<=10;i++){...}
```

- Interface of **scalasca** and **tlog** profiling tools

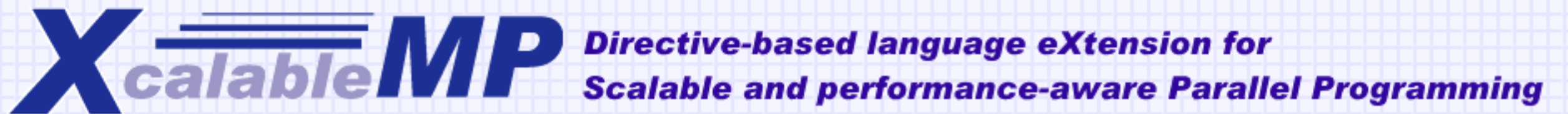


```
#pragma xmp gmove profile
```

```
:
```

```
#pragma xmp loop on t(i) profile
```

# More Information



日本語

[\[home\]](#) [\[publications\]](#) [\[download\]](#) [\[projects\]](#)

## What's XcalableMP

Although MPI is the de facto standard for parallel programming on distributed memory systems, writing MPI programs is often a time-consuming and complicated process. XcalableMP is a directive-based language extension which allows users to develop parallel programs for distributed memory systems easily and to tune the performance by having minimal and simple notations.

The specification is being designed by XcalableMP Specification Working Group which consists of members from academia and research labs to industries in Japan.

The features of XcalableMP are summarized as follows:

- XcalableMP supports typical parallelization based on the data parallel paradigm and work mapping under "global view programming model", and enables parallelizing the original sequential code using minimal modification with simple directives, like OpenMP. Many ideas on "global-view" programming are inherited from HPF (High Performance Fortran).
- The important design principle of XcalableMP is "performance-awareness". All actions of communication and synchronization are taken by directives, different from automatic parallelizing compilers. The user should be aware of what happens by XcalableMP directives in the execution model on the distributed memory architecture.
- XcalableMP also includes a CAF-like PGAS (Partitioned Global Address Space) feature as "local view" programming.
- Extension of existing base languages with directives is useful to reduce rewriting and educational costs. XcalableMP APIs are defined on C and Fortran 95 as a base language.
- For flexibility and extensibility, the execution model allows to combine with explicit MPI coding for more complicated and tuned parallel codes and libraries.
- For multi-core and SMP clusters, OpenMP hybrid programming model.(Under discussion)

XcalableMP is being designed based on the ex

<http://www.xcalablemp.org>

# Summary & Future work

---

- XcalableMP was proposed as a new programming model to facilitate program parallel applications for distributed memory systems
- Omni XcalableMP compiler  
<http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/xcalablemp>
- Future work
  - For accelerators(GPU, etc)
  - Parallel I/O
  - Interface of MPI library, and so on