Productivity and Performance of Global-View Programming with XcalableMP PGAS Language

○Masahiro Nakao, Jinpil Lee,
Taisuke Boku, Mitsuhisa Sato

Center for Computational Sciences, University of Tsukuba
RIKEN Advanced Institute for Computational Science(AICS)

Background

- Partitioned Global Address Space (PGAS) model has been proposed
  - Global address space where any processes can access distributed data transparently
    - Increases development productivity of parallel applications
  - The global address space is logically partitioned between the processes
    - Enables programmers to perform performance-aware parallel programming
  - Two kinds of memory abstract model :
    - Global-view model, Local-view model

Overview of XcalableMP

- XcalableMP(XMP) is a PGAS language http://www.xcalablemp.org
  - Directive-based extension of C99 and Fortran95
  - "Performance-aware" parallel programming (after slide)
  - The basic execution model is SPMD
  - Two memory abstract models in one language :
    - Global-view model
    - Local-view model (compatible with the coarray Fortran)

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

Objective

- XMP global-view model is useful when parallelizing data-parallel programs with minimum code modification
- Consider the **Productivity** and **Performance** of XMP global-view model

How ?

Compare XMP with other PGAS Language

- Unified Parallel C (UPC)
- Global Arrays (GA)
- Coarray Fortran (CAF)
- Chapel
- X10

Why ?

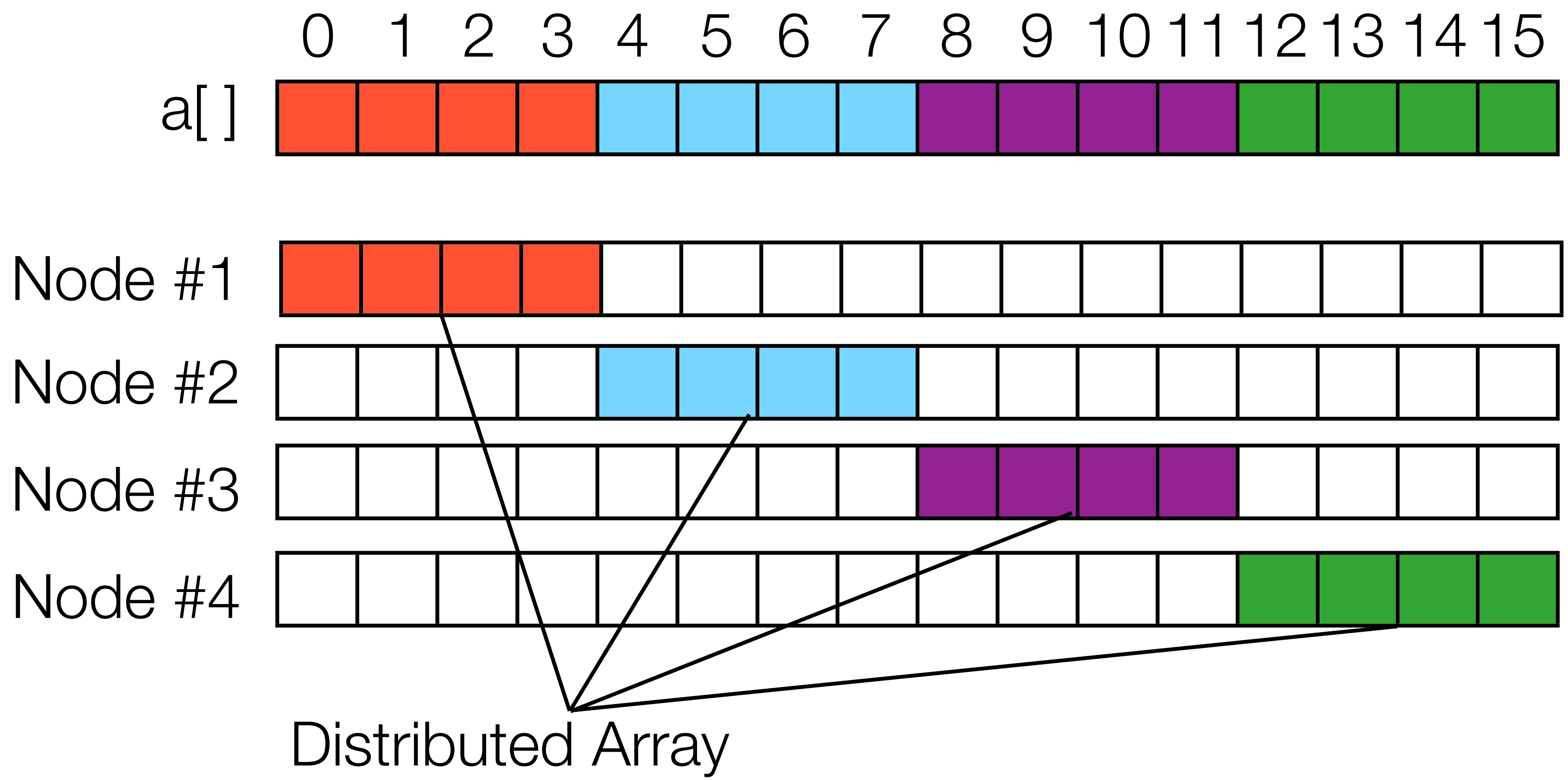
- Global-view model
- C language extension
- SPMD
- Many people use

Outline

- Summarize features of XMP and UPC in global-view model
- Evaluate their Performance and Productivity through some benchmarks

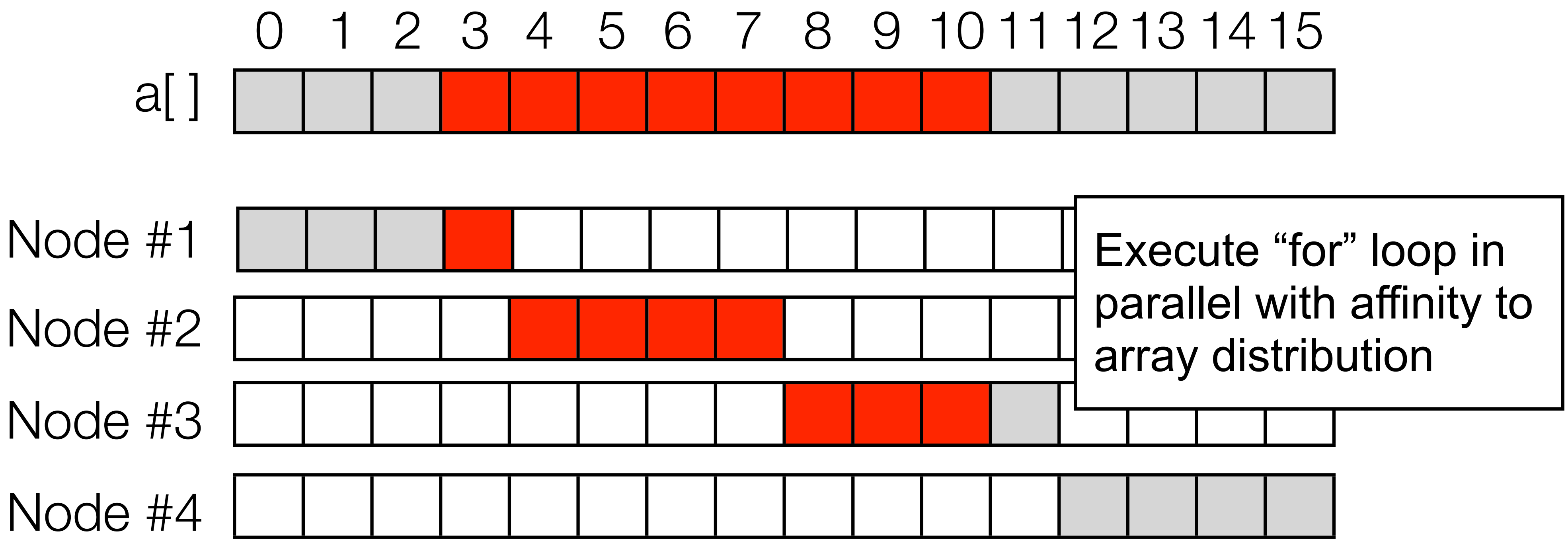
What's Global-view model ?

- Data-mapping and work-mapping automatically
- Example of **data-mapping** :



What’s Global-view model ?

- Data-mapping and work-mapping automatically
- Example of **work-mapping** :



Each node computes Red elements in parallel

Concepts of XMP and UPC

- UPC : Distributed Shared Memory Programming
- XMP : Performance-aware Programming

```
a[i] = tmp;    // a[] is a distributed array, tmp is a local variable
```

- UPC calculates where a[i] is located and its offset
- XMP accesses a[i] directly (no communication)

In XMP, when accessing distributed array with communication, XMP directive should be inserted before the access.

```
#pragma xmp gmove
a[i] = tmp;
```

Because of this policy, XMP may access faster than UPC

Advantage?

- XMP implementation is very simple, but programmer must consider whether needs communication or not
- However, communication points of XMP are more explicit than those of UPC

XMP

```
a[i] = tmp;
```

This line must not occur communication

```
#pragma xmp gmove
a[i] = tmp;
```

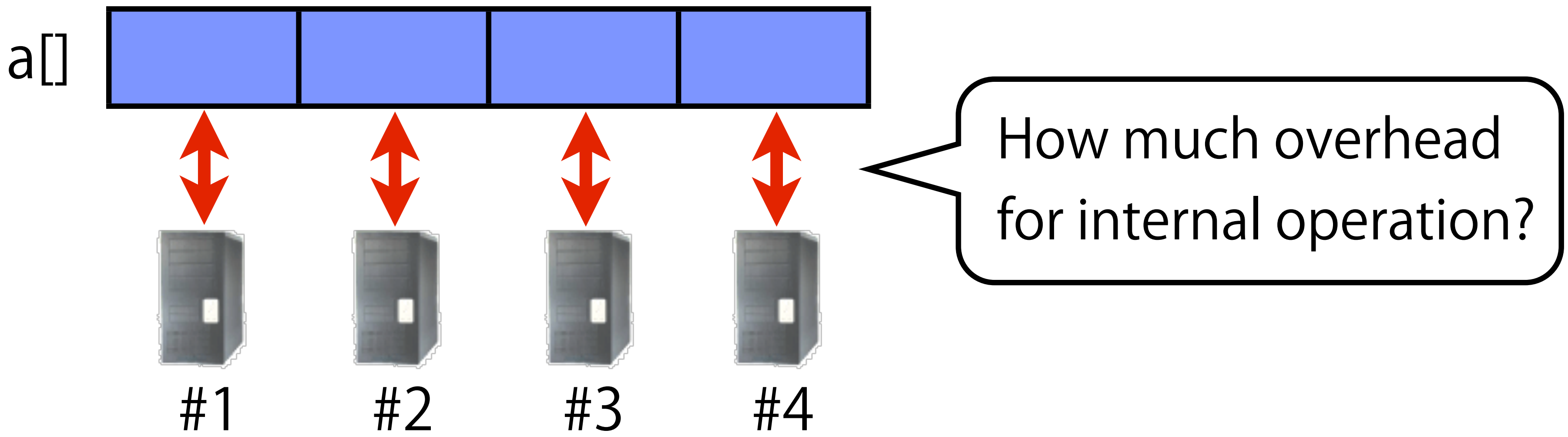
UPC

```
a[i] = tmp;
```

This line may occur communication

Access speed to distributed array

- Evaluate access speed to distributed array, which has an affinity with own process
- Distributed array is accessed in parallel application
  - Access speed is important for its performance



Evaluation of access speed

- Read/write access speed to distributed array within a single node (no-communication)
  - Type array : **double**
  - Number of elements : **2^20** (= 1M) every node
  - Distribution manner : **block, cyclic, block-cyclic**
- Tsukuba Omni XcalableMP Compiler 0.5.4 (TXMP)
- Berkley Unified Parallel C 2.14.0 (BUPC)

XMP

```
#pragma xmp loop on t(i)
for(i = 0; i < N; i++)
   a[i] = tmp;    // tmp is a local
```

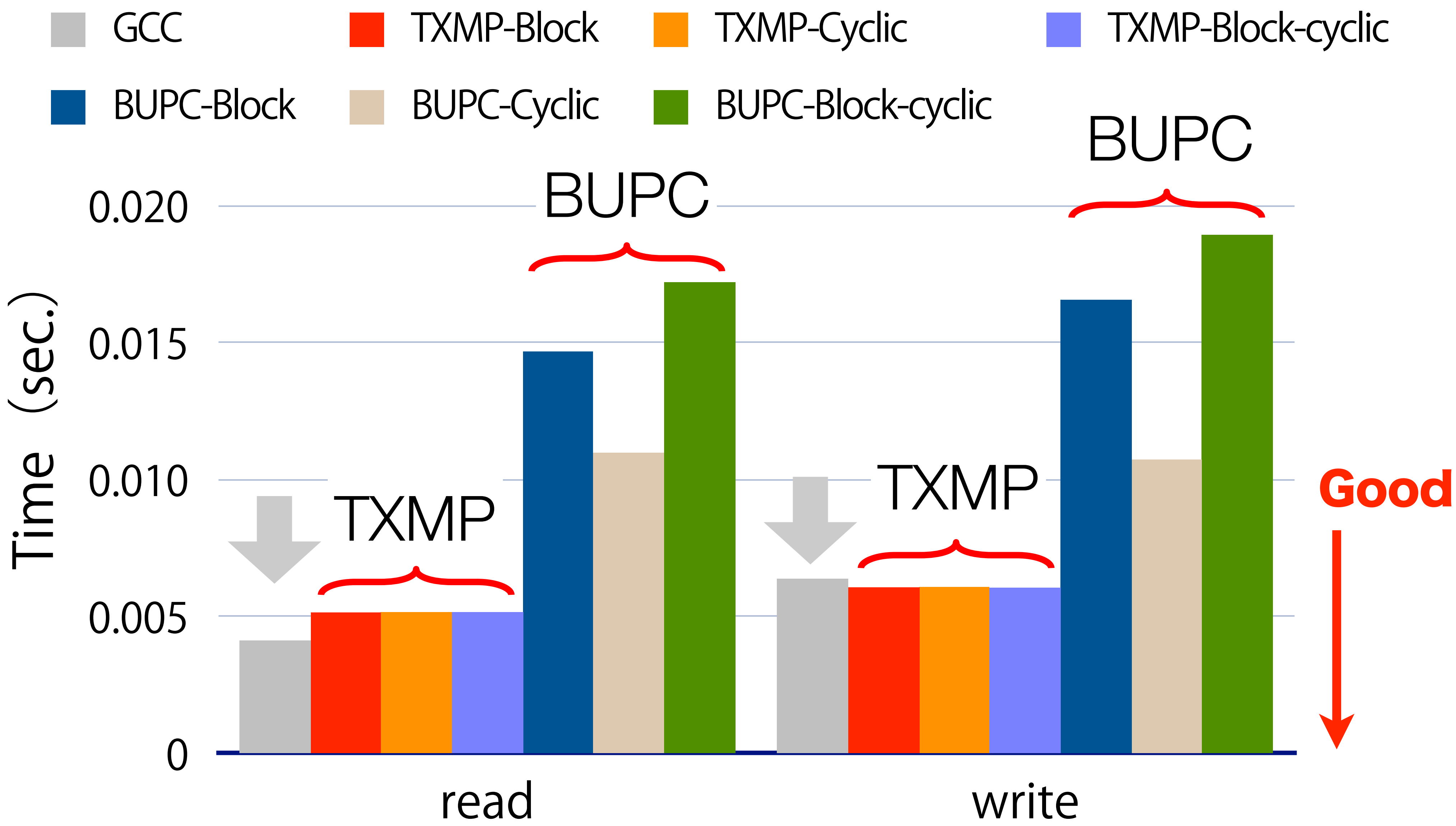
UPC

```
upc_forall(i = 0; i < N; i++; &a[i])
   a[i] = tmp;    // tmp is a local
```

Environment

- T2K Tsukuba System : Linux cluster
  - CPU : AMD Opteron Quad-Core 8356 2.3GHz (4 sockets)
  - Memory : DDR2 667MHz 32GB
  - Network : Infiniband DDR(4rails) 8GB/s

Result


GCC
TXMP-Block
TXMP-Cyclic
TXMP-Block-cyclic
BUPC-Block
BUPC-Cyclic
BUPC-Block-cyclic
Time (sec.)
0.020
0.015
0.010
0.005
0
BUPC
TXMP
BUPC
TXMP
Good
read
write

Discussion

- UPC has a "privatization" technique to speed up for access to distributed array
  - Direct access by using a local address of a distributed array

```
shared double a[SIZE];
double *a_ptr;
a_ptr = &a[MYTHREAD];
   :
for(i=0;i<SIZE/THREADS; i++)
   a_ptr[i] = ....
```

assign a beginning address of distributed array to local pointer

But, program is more complex, because work-mapping must be written by users

XMP can access to distributed array as fast as a backend compiler without "privatization" technique

Outline

- Summarize features of XMP and UPC in global-view model
- Evaluate their Productivity and Performance through some benchmarks

Data layout

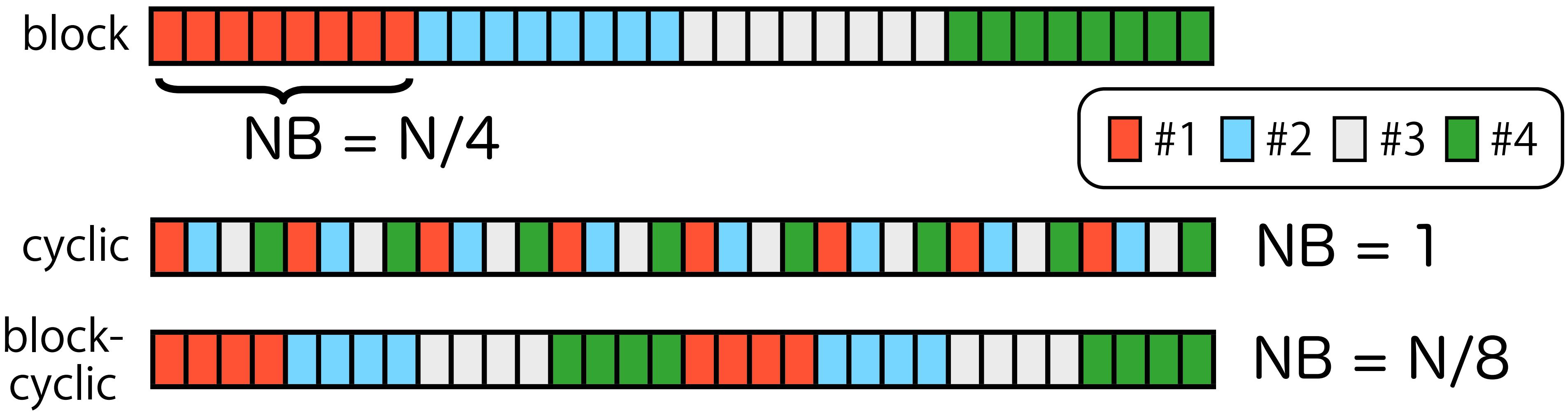
- Data layout is important to
  - Reduce communication and balance CPU loads on each node
  - Adjust any applications

Need to support various data layouts

UPC data distribution

- UPC :

```
shared [NB] double a[N];   // NB is a block size
```



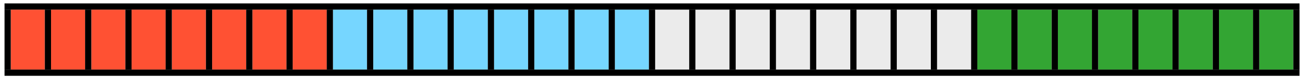
- Merit : Very easy to understand
- Demerit : Only **in order of its memory** (restriction of multi-dimensional array)

XMP data distribution

- The directives specify a data distribution among nodes (inherit from HPF)

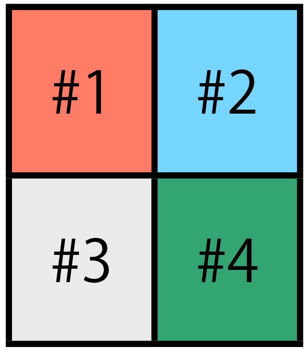
```
double a[N];
#pragma xmp nodes p(4)
#pragma xmp template t(0:N-1)
#pragma xmp distribute t(block) on p
#pragma xmp align a[i] with t(i)
```

block



- Multi-dimensional array is supported

```
double a[N][N];
#pragma xmp nodes p(2, 2)
#pragma xmp template t(0:N-1, 0:N-1)
#pragma xmp distribute t(block, block) on p
#pragma xmp align a[i][j] with t(i,j)
```



Shadow/Reflect directives

- Supports shadow region for stencil applications

```
double a[9];
#pragma xmp nodes p(3)
#pragma xmp template t(0:9)
#pragma xmp distribute t(block) onto p
#pragma xmp align a[i] with t(i)
#pragma xmp shadow a[1:1]          // set width of shadow region
      :       //changing a[]
#pragma xmp reflect (a)            // synchronize shadow region
```

Data distribution

a[9]	#1	#2	#3
	0 1 2	3 4 5	6 7 8

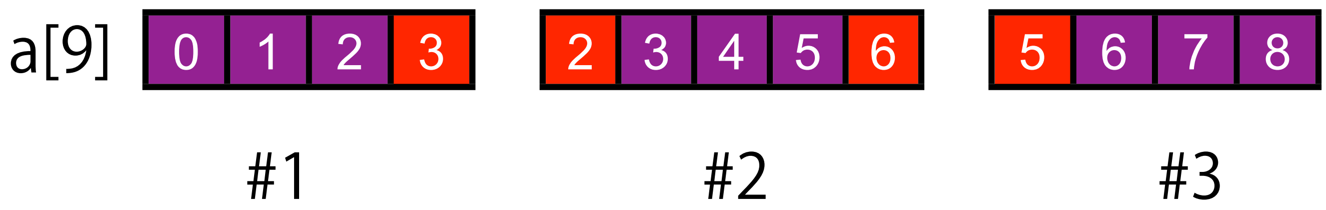
Shadow/Reflect directives

XcalableMP

- Supports shadow region for stencil applications

```
double a[9];
#pragma xmp nodes p(3)
#pragma xmp template t(0:9)
#pragma xmp distribute t(block) onto p
#pragma xmp align a[i] with t(i)
#pragma xmp shadow a[1:1]        // set width of shadow region
    :        //changing a[]
#pragma xmp reflect (a)          // synchronize shadow region
```

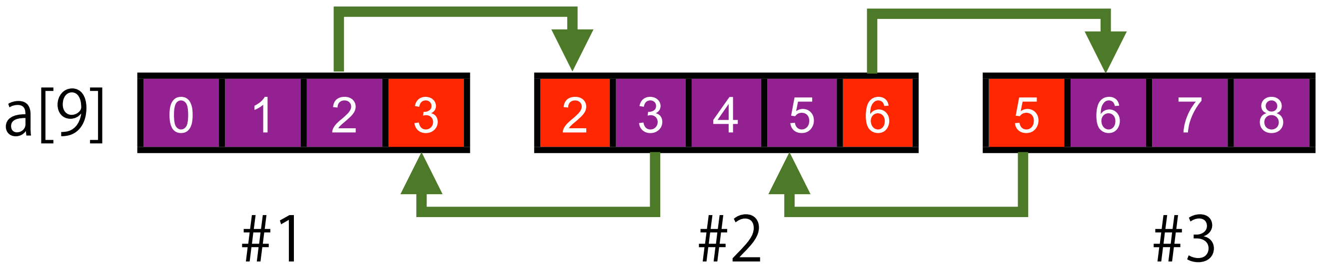
Data distribution



Shadow/Reflect directives

- Supports shadow region for stencil applications

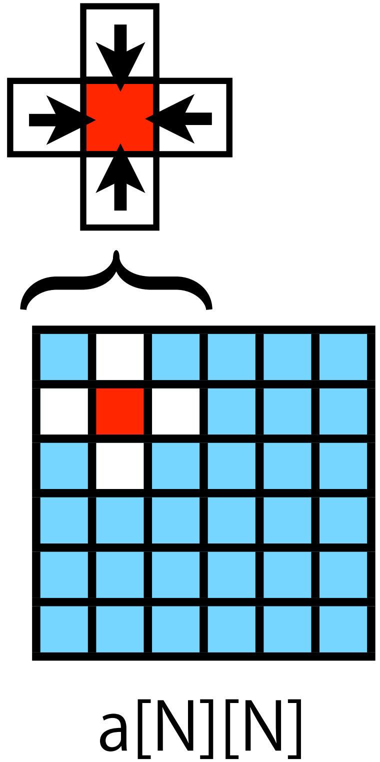
```
double a[9];
#pragma xmp nodes p(3)
#pragma xmp template t(0:9)                   Data
#pragma xmp distribute t(block) onto p        distribution
#pragma xmp align a[i] with t(i)
#pragma xmp shadow a[1:1]        // set width of shadow region
     :      //changing a[]
#pragma xmp reflect (a)          // synchronize shadow region
```



Laplace Solver

- To evaluate the XMP shadow function
- b[y][x] = (a[y+1][x]+a[y-1][x]+a[y][x+1]+a[y][x-1])/4;
- a[][] and b[][] are distributed array

```
#pragma xmp shadow a[1:1][0]   // define shadow
  :
#pragma xmp reflect (a)   // synchronize shadow region
#pragma xmp loop on t(y)
for(y = 1; y < N-1; y++)
   for(x = 1; x < N-1; x++)
      b[y][x] = (a[y-1][x]+a[y+1][x]+a[y][x-1]+a[y][x+1])/4;
```



This XMP code is similar to serial one.

Laplace in UPC

XcalableMP

- In UPC, we use upc_memget() to get shadow region

```
if(THREADS != 1){
  if(MYTHREAD == 0){
    upc_memget(&bottom[1],  &b[WIDTH][1], (N-2)*sizeof(double));
  } else if(MYTHREAD == THREADS-1){
     :
  }
}
upc_barrier;
upc_forall(y=1; y<N-1; y++; &b[y][0]){
  if(MYTHREAD == 0){
    if(y == WIDTH-1){
      for(x=1; x<N-1; x++)   b[y][x] = (a[y-1][x] + bottom[x] + a[y][x-1] + a[y][x+1])/4;
    } else {
        :
```

Needs many if-else statements to communicate and calculate shadow region

- We implemented UPC-privatization version too

Measurement of productivity

XcalableMP

- To measure productivity, we use a Delta SLOC metric[1]
  - The metric indicates how many lines of code change from the original code. How many lines have been modified, added, and deleted from the original code
  - The smaller the total of three metrics or the total source code is, the better the productivity is
  - For example :

Original

```
for(i=0;i<100;i++)
    a[i] = func(i);
```

XMP

Added line : +1

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

UPC

Modified line : +1

```
upc_forall(i=0;i<100;i++;&a[i])
    a[i] = func(i);
```

[1] Andrew I. et. al. , "Evaluating Coarray Fortran with the CGPOP Miniapp", PGAS11, 2011

Productivity

	Original	TXMP	BUPC	BUPC-privatization
Total source code	34	45	73	70
Modified	-	0	4	2
Added	-	11	39	41
Deleted	-	0	0	5
Total delta SLOC	-	11	43	48

Productivity

	Original	TXMP	BUPC	BUPC-privatization
Total source code	34	45	73	70
Modified	-	0	4	2
Added	-	11	39	41
Deleted	-	0	0	5
Total delta SLOC	-	11	43	48

Especially, the value of “Modified” and “Deleted” are 0 !!
This means XMP can parallelize it very simply

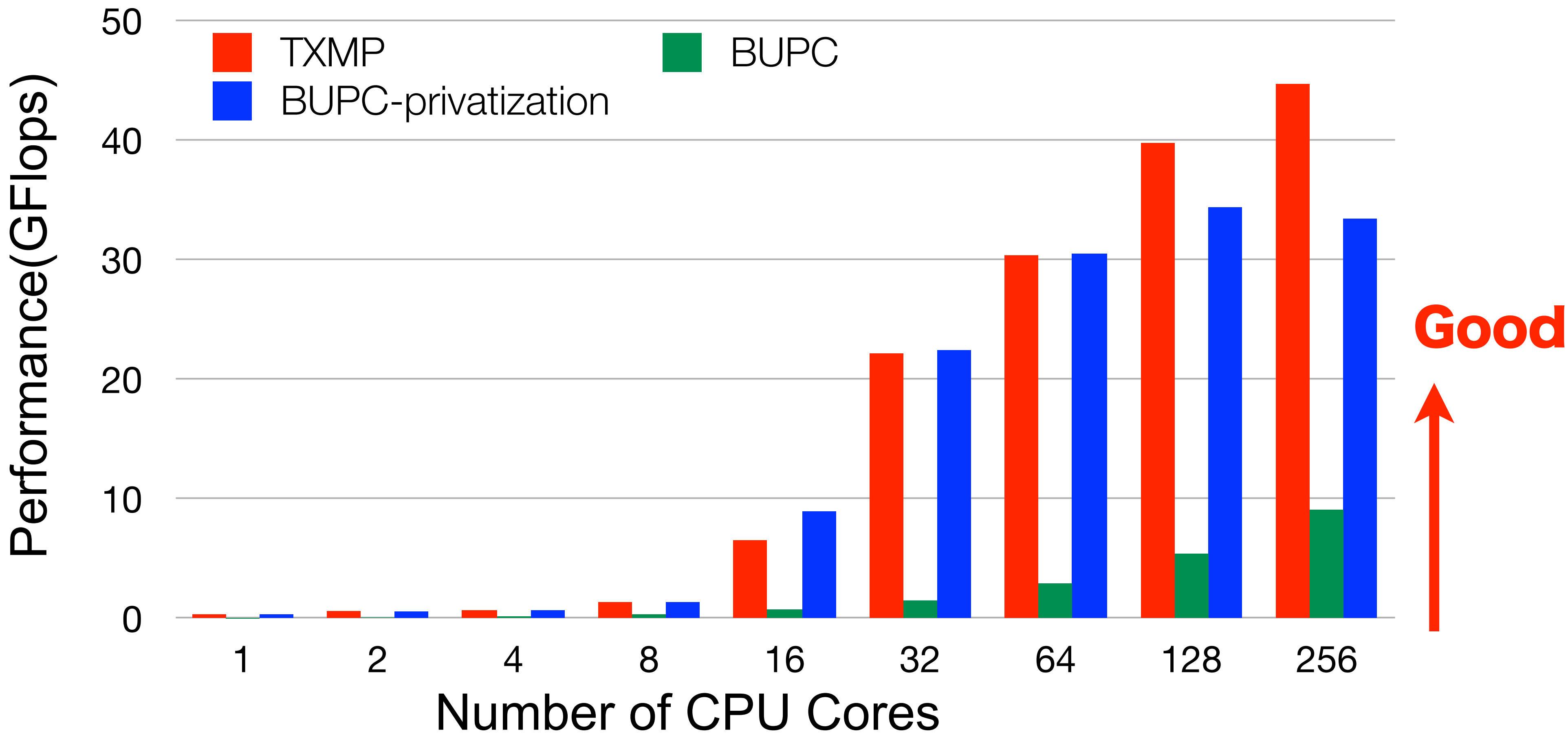
Productivity

	Original	TXMP	BUPC	BUPC-privatization
Total source code	34	45	73	70
Modified	-	0	4	2
Added	-	11	39	41
Deleted	-	0	0	5
Total delta SLOC	-	11	43	48

UPC must use many “if-else” statements for Comm. and Calculation. The productivity of XMP is higher than those of UPCs

Performance

Using array size is 1024x1024, (Strong scaling)



Performance of TXMP is higher than those of BUPCs because there are many “if-else” statements in BUPCs

Conjugate Gradient(CG)

- To evaluate a more general benchmark
- Need to communicate between distributed arrays and local variables for reduction or transposition
- In XMP, we have developed by using 2D process grid and array w[], q[], r[], p[], z[] are distributed
  - Automatically work-mapping
- In UPC, we have used UPC-CG developed by the GWU High-Performance Computing Lab.
  - Only array w[] is distributed
  - Manually work-mapping

http://threads.hpcl.gwu.edu/sites/npb-upc

Conjugate Gradient(CG) XcalableMP

- XMP :
  - When the number of processes is not power-of-two,
    - 2, 8, 32, 128, ...
    - Transferred data is larger than UPC-CG because unused data is reduced by using XMP global-view communication directive
- UPC :
  - Only used data is reduced anytime
    - Each thread calculates beginning and end point of transferred data (is similar to NASA version CG)
    - However, the value of total delta SLOC and total source code are larger than those of XMP-CG

Productivity result

- Implementations of XMP and UPC are based on C language serial CG developed by RWCP in Japan

	Original	TXMP	BUPC	BUPC-privatization
Total source code	376	466	664	651
Modified	-	20	10	3
Added	-	116	296	303
Deleted	-	26	28	28
Total delta SLOC	-	162	334	334

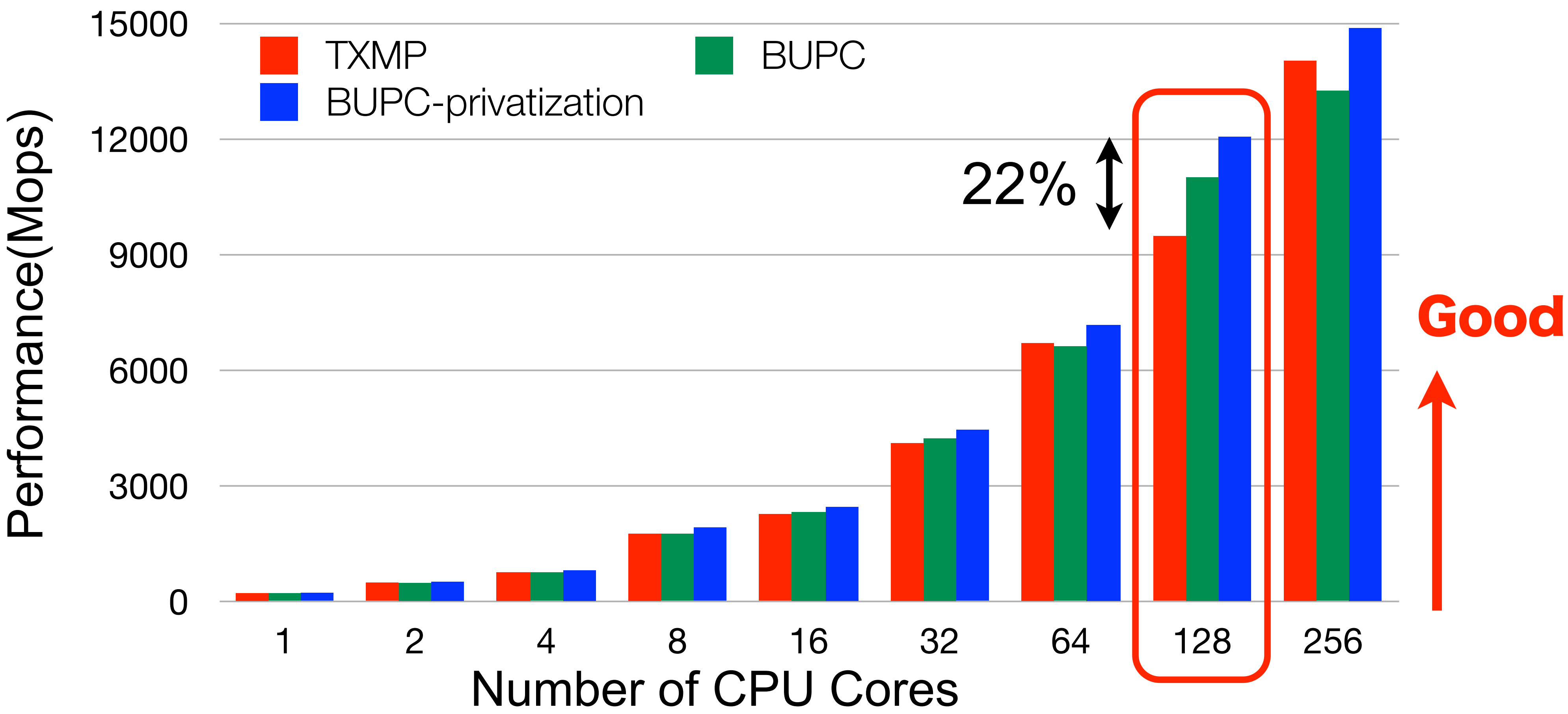
Productivity result

- Implementations of XMP and UPC are based on C language serial CG developed by RWCP in Japan

	Original	TXMP	BUPC	BUPC-privatization
Total source code	376	466	664	651
Modified	-	20	10	3
Added	-	116	296	303
Deleted	-	26	28	28
Total delta SLOC	-	162	334	334

Performance result

Size of array is 150000 x 15000 (Class C), Strong scaling



Summary and Future work

- Summary
  - We investigated productivity and performance of the XMP in global-view model to compare with the UPC
  - XMP supports more data layouts, and has a higher performance access speed to distributed array without “privatization”
  - In laplace solver, the performance and productivity of XMP are higher because XMP supports shadow region
  - In CG, the performance of XMP and UPC is almost the same except 128 CPU cores, the productivity of XMP is high

XMP has a rich global-view programming model that allows it to develop applications with a smaller cost

- Future work
  - Evaluation for real applications in larger number of nodes
  - compare against Chapel, X10, and so on