# XcalableMP and XcalableACC for Productivity and Performance in HPC Challenge Award Competition Class 2 at SC14

Masahiro Nakao[1,2,a)] Hitoshi Murai[1] Hidetoshi Iwashita[1] Takenori Shimosaka[1]
Akihiro Tabuchi[3] Taisuke Boku[2,3] Mitsuhisa Sato[1,2,3]

1. RIKEN Advanced Institute for Computational Science, Japan
2. Center for Computational Sciences, University of Tsukuba, Japan
3. Graduate School of Systems and Information Engineering, University of Tsukuba, Japan
a) masahiro.nakao@riken.jp

## Summary

We present XcalableMP [1–4] implementations of High-performance Linpack (HPL), Fast Fourier Transform (FFT), STREAM, and RandomAccess on the K computer [5]. Moreover, we also present XcalableACC [6, 7] implementations of HPL, FFT, STREAM, and the Himeno Benchmark [8] as an additional benchmark on HA-PACS/TCA [9], which is a GPU cluster.

The highlights of this submission are as follows:

- **Table 1** shows the SLOC (source lines of code) of the implementations.
- **Table 2** shows experimental environments.
- **Table 3** and **Table 4** show performance summaries.

**Table 1** Source lines of code of the implementations

| | HPL | FFT | STREAM | RandomAccess | Himeno |
|---|---|---|---|---|---|
| XcalableMP | 313 | 204 | 69 | 253 | - |
| XcalableACC | 343 | [TBD] | 84 | - | 213 |
| Reference | 8,800 | 787 | 329 | 938 | 365 |

**Table 2** Experimental environments of the K computer and HA-PACS/TCA

| Name | The K computer | HA-PACS/TCA |
|---|---|---|
| Top500/Green500 | 1st / 6th (June, 2011) | 134th / 3rd (November, 2013) |
| CPU & Memory | SPARC64 VIIIfx 2.0 GHz, 8 Cores, 128 GFlops, 1 CPU/Node, DDR3 SDRAM 16 GB, 64 GB/s | Ivy Bridge E5-2680v2 2.8 GHz, 10 Cores, 224 GFlops, 2 CPUs/Node, DDR3 SDRAM 128 GB, 119.4 GB/s (= 59.7 GB/s $\times$ 2 CPUs) |
| Network | Torus fusion six-dimensional mesh/torus network, 5 GB/s x 10 | InfiniBand QDR x 2 rails, 8GB/s |
| Compier & Library | Fujitsu C/Fortran Compiler K-1.2.0-15, Fujitsu MPI K-1.2.0-15, Fujitsu SSLII K-1.2.0-15, FFTE 6.0 | GCC 4.7.2, MVAPICH2 2.0, CUDA 6.5, OpenBLAS 0.2.12, FFTE 6.0 |
| GPU | - | NVIDIA K20X, 1.31/3.95 TFlops (DP/SP), GDDR5 SDRAM 6 GB, 250 GB/s, 4GPUs/Node |

**Table 3** Performance summary of XcalableMP on the K computer

| Benchmark | #Nodes | Performance (/peak) |
|---|---|---|
| HPL | 16,384 | 970.97 TFlops (46.30%) |
| FFT | 36,864 | 79.45 TFlops (1.68%) |
| STREAM | 82,944 | 3,582.50 TB/s (67.49%) |
| RandomAccess | 16,384 | 254.20 GUPS (-) |

**Table 4** Performance summary of XcalableACC on HA-PACS/TCA

| Benchmark | #Nodes (GPUs) | Performance (/peak) |
|---|---|---|
| HPL | 32 (128 GPUs) | 7,102.36 GFlops (4.22%) |
| FFT | [TBD] | [TBD] |
| STREAM | 32 (128 GPUs) | 7,238.10 GB/s (20.21%) |
| Himeno | 32 (128 GPUs) | 6,870.98 GFlops (1.36%) |

The remainder of the present paper is structured as follows. Section I introduces XcalableMP and XcalableACC programming models. Section II describes an evaluation environment. Section III presents implementations and performances of benchmarks using XcalableMP on the K computer. Section IV presents implementations and performances of benchmarks using XcalableACC on HA-PACS/TCA. Section V summarizes the present paper.

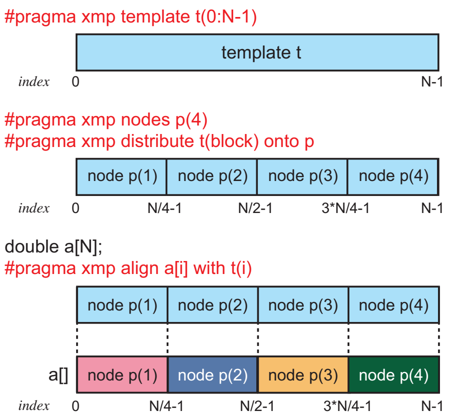
## 1. Proposed Programming Models

### 1.1 XcalableMP

XcalableMP [1–4], XMP for short, is a directive-based language extension for distributed memory systems, which has been designed by XMP specification working group of the PC cluster consortium [10]. It allows users to develop parallel applications and to tune its performance easily. A part of the design is based on experiences of High Performance Fortran [11, 12] and Coarray Fortran [13].

The features of XMP are as follows:

- XMP supports typical parallelization under "global-view memory model," and enables parallelizing the original sequential code using minimal modification with simple directives.
- XMP also includes coarray features for "local-view memory model."
- XMP is defined as an extension for familiar languages, such as C and Fortran, to reduce code-rewriting and educational costs.
- XMP is defined to mix XMP and OpenMP directives for multi-thread programming.



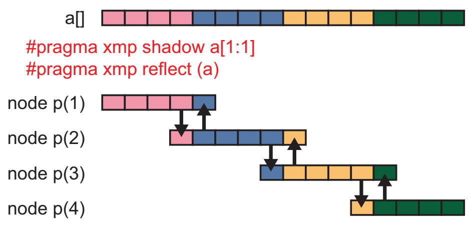
```
#pragma xmp template t(0:N-1)

#pragma xmp nodes p(4)
#pragma xmp distribute t(block) onto p

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

template t
index 0 N-1
node p(1) node p(2) node p(3) node p(4)
index 0 N/4-1 N/2-1 3*N/4-1 N-1
node p(1) node p(2) node p(3) node p(4)
a[] node p(1) node p(2) node p(3) node p(4)
index 0 N/4-1 N/2-1 3*N/4-1 N-1

**Fig. 1** Definition of a distributed array in XcalableMP [3]

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

**Fig. 2** Loop parallelization in XcalableMP [3]

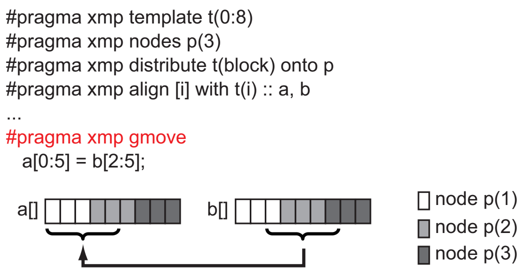


```
#pragma xmp shadow a[1:1]
#pragma xmp reflect (a)
```

**Fig. 3** Definition and synchronization of a halo region in XcalableMP [3]

XMP defines a distributed array by using a virtual index set called a "template." **Fig. 1** and **Fig. 2** show an example of an XMP distributed array and parallelization of a loop statement. A **template** directive defines a template *t*, the virtual indexes of which are from 0 to N - 1. A **node** directive defines a node set *p*, which consists of four nodes. A **distribute** directive distributes the template *t* onto the node *p* in a *block* manner, which means the same number of blocks are distributed. XMP also provides *cyclic*, *block-cyclic*, and *user-defined* distributions. A **align** directive defines the distributed array *a[]* and aligns it with the template *t*. Each node allocates reasonable memory for the distributed array *a[]*. In Fig. 2, a **loop** directive parallelizes a loop statement depending on the template *t*. For example, if N is 16, node *p(1)* handles the iterations indexes from 0 to 3. A that time, the OpenMP **parallel** directive also parallelizes the loop statement parallelized by the XMP directive. In this case, the order of the XMP **loop** directive and the OpenMP **parallel** directive does not matter.

In order to develop stencil applications easily, which are most commonly found in computer simulations, XMP provides **shadow** and **reflect** directives. These directives can be used to define a halo region for a distributed array and synchronize it between neighborhood nodes. Fig. 3 shows an example of how to use the directives. A **shadow** directive defines a halo region in both sides of a distributed array *a[]*. A **reflect** directive synchronizes data of the defined halo region onto the neighborhood nodes.



```
#pragma xmp template t(0:8)
#pragma xmp nodes p(3)
#pragma xmp distribute t(block) onto p
#pragma xmp align [i] with t(i) :: a, b
...
#pragma xmp gmove
  a[0:5] = b[2:5];
```

**Fig. 4** XcalableMP **gmove** directive [3]

```
#pragma xmp bcast (e) from p(2)
```

**Fig. 5** XcalableMP **bcast** directive

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

**Fig. 6** XcalableMP **reduction** directive

```
array_name[start:length]:[node_num]
```

**Fig. 7** XcalableMP **coarray** feature

Besides, XMP provides various communication directives. In **Fig. 4**, a **gmove** directive enables programmers to access distributed arrays using global indexing, where five elements from *b[2]* to *b[6]* are copied to five elements from *a[0]* to *a[4]*. In **Fig. 5**, a **bcast** directive performs broadcast communication from a specified node, where the local variable *e* stored on node *p(2)* is broadcasted. In **Fig. 6**, A **reduction** directive performs a reduction operation among nodes, where the **reduction** directive calculates the total value of the local variable *f* stored on all nodes. **Fig. 7** is an example of coarray in XMP C language. Fig. 7 means *length* elements from *array_name[start]* to *array_name[start+length-1]* located on a node specified by *node_num* are referred.

## 1.2 XcalableACC

XcalableACC [6,7], XACC for short, enables programmers to mix XMP and OpenACC [14] directives in order to develop applications that can use accelerator clusters with ease. In XACC, programmers can use both XMP and OpenACC directives seamlessly. Additionally, XACC also provides directives to transfer data among accelerator memories directly.

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

**Fig. 8** Loop parallelization in XcalableACC

```
 1  double a[N];
 2  #pragma xmp template t(0:N−1)
 3  #pragma xmp nodes p(4)
 4  #pragma xmp distribute t(block) onto p
 5  #pragma xmp align a[i] with t(i)
 6  #pragma xmp shadow a[1:1]
 7  ...
 8  #pragma acc data copy(a)
 9  {
10    for(int n=0;n=NITER;n++){
11      ...
12  #pragma xmp reflect (a) acc
13  #pragma xmp loop on t(i)
14  #pragma acc parallel loop
15      for(int i=1;i<N−1;i++){
16        ... = a[i−1] + a[i+1];
```

**Fig. 9** Definition and synchronization of a halo region using **shadow** and **reflect** directive in XcalableACC

```
 1  double a[N];
 2  #pragma xmp template t(0:N−1)
 3  #pragma xmp nodes p(4)
 4  #pragma xmp distribute t(block) onto p
 5  #pragma xmp align a[i] with t(i)
 6  #pragma xmp shadow a[1:1]
 7  ...
 8  #pragma acc data copy(a)
 9  {
10  #pragma xmp reflect_init (a) acc
11    for(int n=0;n=NITER;n++){
12      ...
13  #pragma xmp reflect_do (a) acc
14  #pragma xmp loop on t(i)
15  #pragma acc parallel loop
16      for(int i=1;i<N−1;i++){
17        ... = a[i−1] + a[i+1];
```

**Fig. 10** Definition and synchronization of a halo region using **shadow**, **reflect_init**, and **reflect_do** directives in XcalableACC

Fig. 8 shows an example of a loop parallelization in XACC. In lines 1 to 5, XMP directives define the distributed array *a[]*. This process is the same as XMP programming model in Fig. 1. In line 7, the OpenACC **data** directive transfers the distributed array *a[]* to accelerator memory. In line 9, the XMP **loop** directive parallelizes the loop on each node. At that time, the OpenACC **parallel** directive with **loop** clause in line 10 also parallelizes the loop statement parallelized by the XMP directives. In this case, the order of the XMP **loop** directive and the OpenACC **parallel** directive does not matter.

XACC also provides directives to transfer data among accelerator memories directly by adding **acc** clause to existing XMP communication directives. **Fig. 9** shows an example of an exchange a halo region. In line 12, the **reflect** directive with **acc** clause performs synchronization the halo region on accelerator memory among neighborhood nodes.

Besides, the **reflect** directive is often called repeatedly with the same condition in a loop statement. In addition, the **reflect** directive internally performs some initialization, for example, to set neighborhood nodes and to calculate offsets for data transfers. However, the initialization only needs to be done once. In order to remove the unnecessary multiple initializations, we propose dividing the **reflect** directive into two directives, an **reflect_init** directive and an **reflect_do** directive, similar to MPI persistent communication. The **reflect_init** directive performs the initialization, and the **reflect_do** directive performs communication. Of course, this division will also be effective for the **reflect** directive for host data. **Fig. 10** shows an example using the **reflect_init** and **reflect_do** directives. In line 10, the **reflect_init** directive with **acc** clause performs initialization the halo region of the array *a[]* on accelerator memory. In line 13, the **reflect_do** directive with **acc** clause performs synchronization the halo region among neighborhood nodes.

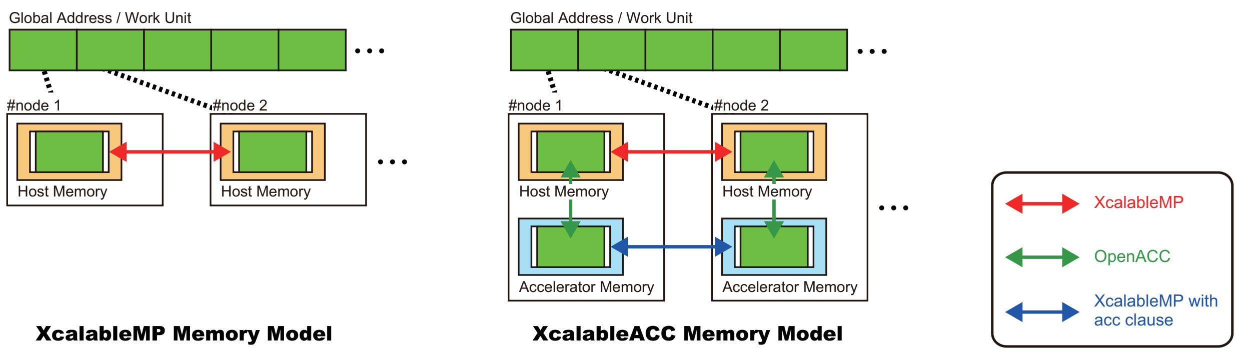


**Fig. 11** XcalableMP and XcalableACC memory models

**Fig. 11** shows XMP and XACC memory models. In XACC memory model, XMP directives are used to transfer data among host memories, OpenACC directives are used to transfer data between host memory and accelerator memory, and XMP directives with **acc** clause are used to transfer data among accelerator memories directly.

## 2. Evaluation Environment

### 2.1 Omni Compiler

We have been developing the Omni compiler [15] that supports XMP, OpenACC, and OpenMP directives. The Omni compiler translates C or Fortran source code with these directives and generates parallel code. The Omni compiler is available at `http://omni-compiler.org` as an open-source software.

### 2.2 Machines

In order to evaluate the performance of HPCC benchmarks, we used two machines: the K computer [5] (**Fig. 12**) and HA-PACS/TCA [9] (**Fig. 13**). HA-PACS/TCA is equipped with the NVIDIA K20X GPU. While the K computer is used to evaluate the XMP programming model, HA-PACS/TCA is used to evaluate the XACC programming model. Table 2 shows the specification of the two machines. Additionally, **Fig. 14** shows the detail of the computation node of HA-PACS/TCA. Each computation node has two CPUs and four GPUs. The memory size of each GPU is 6 GB, and the memory size of host memory is 128GB. The bandwidth of each GPU GDDR5 memory is 250 GB/s, the bandwidth between GPU memory and host memory is 8GB/s, and the bandwidth between host memory and CPU is 119.4 GB/s (= 59.7 GB/s × 2 CPUs).

**Fig. 12** The K computer [16]

**Fig. 13** HA-PACS/TCA [9]

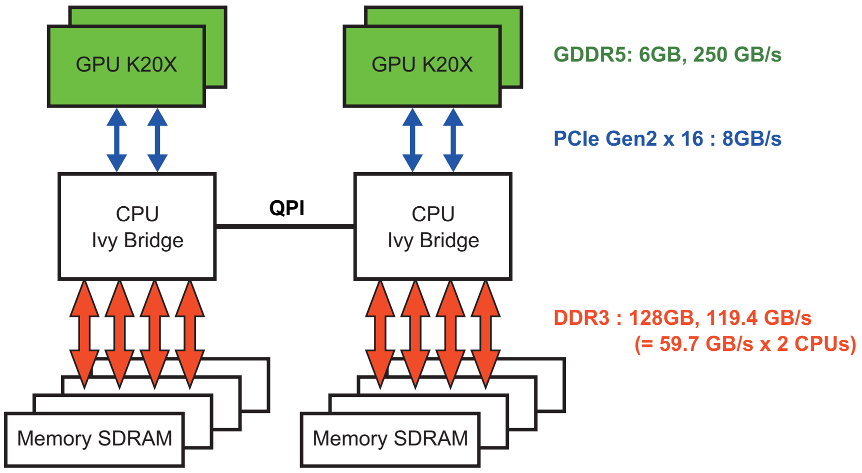


**Fig. 14** Computation node in HA-PACS/TCA

### 2.3 How to count source lines of code

Table 1 summarizes the source lines of code (SLOC) of all implementations. All SLOC is excluded comments and blank lines, but included a validation operation and a printing performance result.

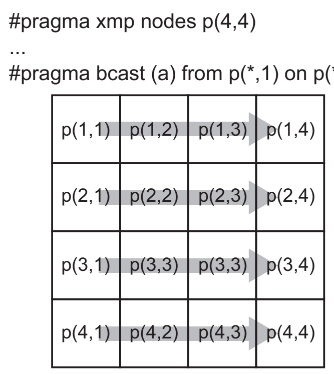
# 3. Implementation and Performance of benchmarks using XcalableMP on the K computer

## 3.1 High-performance Linpack in XcalableMP

### 3.1.1 Overview

We implemented an HPL algorithm written in XMP C language. The SLOC is **313**.

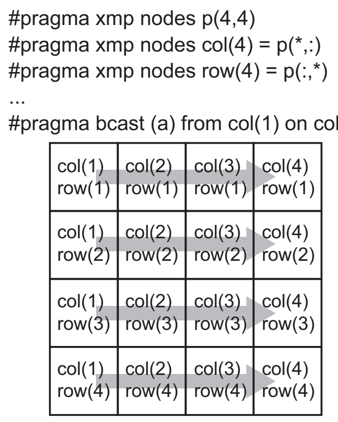
### 3.1.2 Differences from the implementation of last year



```
#pragma xmp nodes p(4,4)
...
#pragma bcast (a) from p(*,1) on p(*,:)
```

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

**Fig. 15** Broadcast operation using two-dimensional node set (Last year)



```
#pragma xmp nodes p(4,4)
#pragma xmp nodes col(4) = p(*,:)
#pragma xmp nodes row(4) = p(:,*)
...
#pragma bcast (a) from col(1) on col(:)
```

col(1) row(1) col(2) row(1) col(3) row(1) col(4) row(1)
col(1) row(2) col(2) row(2) col(3) row(2) col(4) row(2)
col(1) row(3) col(2) row(3) col(3) row(3) col(4) row(3)
col(1) row(4) col(2) row(4) col(3) row(4) col(4) row(4)

**Fig. 16** Broadcast operation using split node set (This year)

Last year, we used a two-dimensional node set *p* to perform broadcast operation as **Fig. 15**. While the "*" means duplicate executing, the ":" represents all node in the dimension. For example, node *p(1,1)* transfers data to only own row nodes: *p(1,2)*, *p(1,3)*, and *p(1,4)*. However, the XMP runtime internally creates new communicators using MPI_Comm_split() every time in the **bcast** directive. In order to remove the unnecessary MPI_Comm_Split() operations, we create one-dimensional node set *col* and *row* from the two-dimensional node set *p* in advance. **Fig. 16** shows an example of new broadcast operation by using the split node set. For example, *col(1)* consists of *p(1,1)*, *p(2,1)*, *p(3,1)*, and *p(4,1)*. In Fig. 16, the all *col(1)* transfer data to own row nodes. The node set *row* is also used to transfer data to only own column nodes. Thus, the node sets *col* and *row* are very useful to implement an HPL algorithm.

### 3.1.3 Implementation

□ Block-cyclic distribution

A coefficient matrix *A[][]* is distributed to each node in a block-cyclic manner, as hpcc-1.4 HPL. In the following code, a **template** directive declares a two-dimensional template *t*, and a **node** directive declares a two-dimensional node set *p*. A **distribute** directive distributes the template *t* onto $P \times Q$ nodes in the same block size where *NB* is the block size. Finally, an **align** directive aligns *A[][]* with the template *t*. **Fig. 17** shows the block-cyclic distribution in the following code. Note that the order of the template indexes is based on Fortran conventions. Therefore, in XMP C language, the order of *i* and *j* in **align** directive is reversed, shown in line 5 and Fig. 17.

```
1  double A[N][N];
2  #pragma xmp template t(0:N−1, 0:N−1)
3  #pragma xmp nodes p(P,Q)
4  #pragma xmp distribute t(cyclic(NB), cyclic(NB)) onto p
5  #pragma xmp align A[i][j] with t(j,i)
```

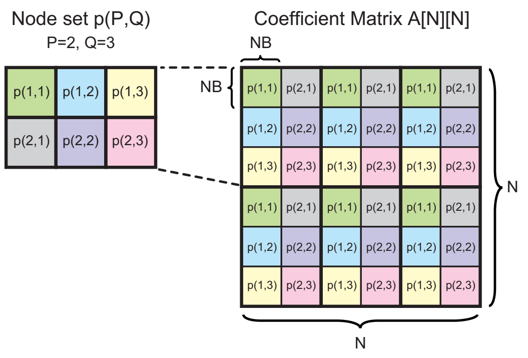


**Fig. 17** Block-cyclic distribution

□ Swap operation by using split node set

Swap operation, exchanging a pivot row, is used by split node set, described in Section 3.1.2 as following.

```
#pragma xmp bcast (pivot_row) from col(get_col_key(target)) on col(:)
```

□ Panel broadcast by using **gmove** directive

The following code and **Fig. 18** indicate a panel broadcast operation using **gmove** directive. The array *A_L[][]* is also distributed in the block-cyclic manner, but only the first dimension of the array *A_L[][]* is distributed. Thus, target elements of the array *A[j+NB:N-j-NB][j:NB]* (stripe block in Fig. 18) are broadcast to the array *A_L[j+NB:N-j-NB][0:NB]* that exists on each node.

```
double A[N][NB];
#pragma xmp align A_L[i][*] with t(*,i)
  :
#pragma xmp gmove
  A_L[j+NB:N−j−NB][0:NB] = A[j+NB:N−j−NB][j:NB];
```

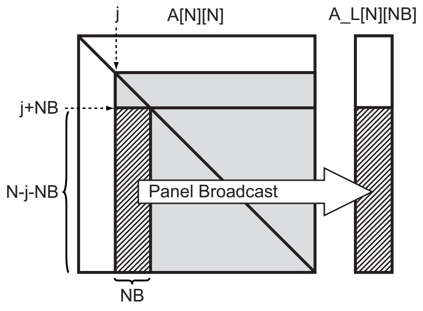


**Fig. 18** Panel broadcast

□ Usage of BLAS library for distributed array

High performance mathematical libraries, for example BLAS and ScaLAPACK, are often used in computational science. In XMP, a programmer can use these libraries for a distributed array. XMP has a rule that a pointer of an XMP distributed array indicates a local pointer. The following code shows that *DGEMM* function applies distributed arrays *A[][]*, *A_L[][]*, and *A_U[][]*. Note that when using mathematical libraries, a programmer may need some information of a local array, for example leading dimension. Thus, XMP provides useful functions to get information of a local array from a distributed array. The function **xmp_array_lead_dim()** returns a leading dimension of a local array. We believe that mixing global-view and local-view gives programmers good productivity with high performance.

```
xmp_desc_t A_desc = xmp_desc_of(A); // Get descriptor of distributed array
int ld[2];
xmp_array_lead_dim(A_desc, ld);
int ld_A = ld[1];
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, local_len_y, local_len_x, NB, −1.0,
            &A_L[global_y][0], ld_A_L, &A_U[0][global_x], ld_A_U, 1.0, &A[global_y][global_x], ld_A);
```

### 3.1.4 Performance

We performed the HPL with eight threads per process on one node. The size of the coefficient matrix *A[][]* is about 70% of the system memory. **Table 5** shows the performance and the theoretical peak performance of the system. For comparison, Table 5 also shows the performance of the HPL of last year. The best performance is **970.97** TFlops for 16,384 nodes. The performance of this year is a little better than that of last year thanks to the split node set described in Section 3.1.2.

**Table 5** Performance of HPL in XcalableMP

| #Nodes | Performance (TFlops) | |
|---|---|---|
| | **XcalableMP of this year (/peak)** | XcalableMP of last year (/peak) |
| 1 | 0.11 (82.52%) | 0.10 (77.86%) |
| 4 | 0.40 (78.83%) | 0.38 (75.32%) |
| 16 | 1.58 (76.99%) | 1.46 (73.13%) |
| 64 | 6.19 (75.55%) | 5.70 (71.26%) |
| 256 | 23.72 (72.38%) | 21.95 (68.58%) |
| 1,024 | 88.12 (67.23%) | 81.37 (63.57%) |
| 4,096 | 309.64 (59.06%) | 286.20 (55.90%) |
| 16,384 | 970.97 (46.30%) | 933.80 (44.50%) |

### 3.1.5 More scaling improvement

We have been implementing a new HPL which uses asynchronous **gmove** directive to implement "look-ahead" algorithm. The following code shows a part of the code. If the **async** clause is specified in XMP directive, the statements following the directive may be executed while the operation continues asynchronously. To guarantee the asynchronous operation is complete, the **wait_async** directive is used. We believe that this operation will improve the performance. If we finish the implementation by SC14, we hope to have a chance to present this result at BoF in SC14.

```
#pragma xmp gmove async (1)
  A_L[j+NB:N−j−NB][0:NB] = A[j+NB:N−j−NB][j:NB];

...
#pragma xmp wait_async (1)
```

## 3.2 Fast Fourier Transform in XcalableMP

### 3.2.1 Overview

We implemented an FFT algorithm written in XMP Fortran calling FFTE [17] library routine zfft1d. The SLOC is **204**.

### 3.2.2 Differences from the XMP implementation of last year

Last year, we ported the hpcc-1.4 FFT into XMP. It still contained a few MPI calls and included C programs to describe inter-procedure interfaces. Thus, the SLOC was too large. In this year, we have rewritten it only in XMP Fortran. Thanks to the powerful and natural interprocedure-interfaces of XMP, the program became very simple and easy to read. Moreover, in order to tune its performance and to simplify the program, we changed how to use OpenMP thread parallelism. While calling FFTE library in each OpenMP thread last year, calling OpenMP version FFTE in each process this year.

The XMP **nodes** directive uses MPI_Comm_dup() in Omni XMP compiler. In case of more than ten thousand nodes, the MPI_Comm_dup() takes large cost. Thus, we move the XMP **nodes** directive from a local function to module common in the XMP FFT. Moreover, we modified the Omni XMP compiler to reduce the number of calling MPI_Comm_split(). While MPI_Comm_split() was always called for every XMP **align** directive in last year, it is quite rarely called in the latest Omni XMP compiler.

### 3.2.3 Implementation

The XMP FFT consists of three files: "main.f90," "xmp-fft.f90," and "xmp-zfft1d.f90."

The global FFT algorithm of HPCC is described in "xmp-fft.f90." In this file, subroutine xmpfft is called from the main and gets suitable combination of sizes **na** and **nb**, where **na** × **nb** = **n**, **n** is the length of vector. Subroutine xmfft1 is called from xmpfft and allocates each part of huge global arrays **a**, **b** and **w** for each process using automatic array feature in Fortran90. The size is globally **n** and locally **n** / (number of nodes) for each.

The six-step FFT algorithm called from the global FFT is described in "xmp-zfft1d.f90." Code fragments in the file is shown in bellow. As dummy arguments of subroutine xmpzfft1d, the objects of global two-dimensional arrays **a**, **b** and **w** are inherited from the caller xmpfft1 without any extra overhead. The **template**, **distribute**, and **align** directives describe block distribution attributes of arrays. In the six-step FFT, global matrix transposition must be performed three times. The Omni XMP compiler provides an effective global transpose library "xmp_transpose()," as shown in **Fig. 19**, which can be called simply with XMP distributed arrays as actual arguments. Also outside of the kernel part, XMP program usefully accepts OpenMP pragmas. The last code fragment shows combined parallelization of XMP and OpenMP for the outermost loop **do** *i*.

```
subroutine xmpzfft1d(a, b, c, w, n, nx, ny, c_size, is_back)
...
  complex*16 a(nx,ny), b(ny,nx), w(ny,nx)
!$xmp template tx(nx)
!$xmp template ty(ny)
!$xmp distribute tx(block) onto p
!$xmp distribute ty(block) onto p
!$xmp align a(*,i) with ty(i)
!$xmp align b(*,i) with tx(i)
!$xmp align w(*,i) with tx(i)
```

```
!$xmp loop on tx(i)
!$omp parallel do
do i=1,nx
  do j=1,ny
    b(j,i)=b(j,i)*w(j,i)
  end do
end do
```

```
call xmp_transpose(b, a, 1)    ! a and b are XMP distributed arrays
```

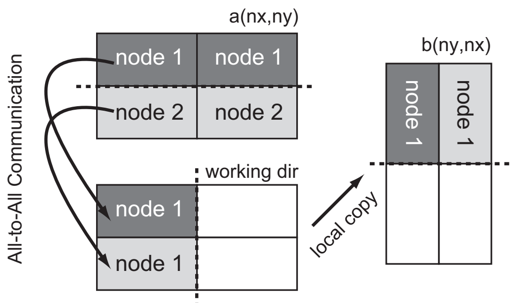


**Fig. 19** Behavior of the xmp_transpose()

### 3.2.4 Performance

We performed the XMP FFT with eight threads per process on one node. In order to use effectively cache on the K computer, we set parameters, the number of nodes and length of vector [18]. The length of vector occupies about 35 % of system memory. For comparison, we also evaluated the modified hpcc-1.4 FFT which is optimized for the K computer. **Table 6** shows the results where the best performance of the XMP FFT is **79.45 TFlops** for 36,864 nodes. The performance of the XMP FFT is almost the same as that of the hpcc-1.4 FFT.

**Table 6** Performance of FFT in XcalableMP

| #Nodes | Performance (TFlops) | | |
|---|---|---|---|
| | **XcalableMP of this year (/peak)** | XcalableMP of last year (/peak) | modified hpcc-1.4 (/peak) |
| 36 | **0.09 (1.89%)** | 0.13 (2.79%) | 0.14 (2.91%) |
| 144 | **0.55 (2.97%)** | 0.64 (3.47%) | 0.68 (3.67%) |
| 576 | **2.13 (2.89%)** | 2.14 (2.90%) | 2.21 (3.00%) |
| 2,304 | **8.27 (2.80%)** | 8.16 (2.77%) | 8.43 (2.86%) |
| 9,216 | **27.72 (2.35%)** | 25.26 (2.14%) | 25.55 (2.17%) |
| 36,864 | **79.45 (1.68%)** | 50.08 (1.08%) | 72.82 (1.56%) |

## 3.3 STREAM in XcalableMP

### 3.3.1 Overview

We implemented a STREAM algorithm written in XMP C language. The SLOC is **69**.

### 3.3.2 Differences from the XMP implementation of last year

We developed two XMP STREAM benchmarks last year. One was flat-MPI version, which was in the submission report last year. The other was multi-threaded version, which was in the presentation of BoF at SC13. In this year, we add "**#pragma loop xfill**" and "**#pragma loop noalias**" directives to the multi-threaded version, which are optimization directives of the Fujitsu compiler. While "**#pragma loop xfill**" ensures one cache line to store write-only data, "**#pragma loop noalias**" indicates that there is no possibility that the different pointer variables do not indicate the same storage area. Of cause, the directives are useful for the hpcc-1.4 STREAM.

### 3.3.3 Implementation

The following code shows a part of an XMP STREAM code. The program is quite straightforward. Basically, a programmer only adds XMP directives into the sequential STREAM benchmark. Line 1 defines XMP node set *p* to parallelize this program. Line 3 to 7 is a main kernel of STREAM. Line 9 performs a reduction operation among nodes to calculate a total performance.

```
1  #pragma xmp nodes p(*)
2  ...
3  #pragma loop xfill
4  #pragma loop noalias
5  #pragma omp parallel for
6    for (j=0; j<size; j+=1)
7      a[j] = b[j] + scalar*c[j];
8  ...
9  #pragma xmp reduction(+:triadGBs)
```

### 3.3.4 Performance

We performed the STREAM with eight threads per process on one node. The vector lengths of the arrays *a[]*, *b[]*, *c[]* is 536,870,912 which occupies 75 % of the system memory. For comparison, we also evaluated the hpcc-1.4 STREAM benchmark with the"**#pragma loop xfill**" and "**#pragma loop noalias**" directives. **Table 7** shows the performances, and also shows the performances of the multi-threaded version STREAM of last year. The best performance of the XMP STREAM is **3,583 TB/s** for 82,944 nodes (full node). The performance of the XMP STREAM is as the same as that of hpcc-1.4 with the directives.

**Table 7** Performance of STREAM in XcalableMP

| #Nodes | Performance (GB/s) | | |
|---|---|---|---|
| | **XcalableMP of this year (/peak)** | hpcc-1.4 (/peak) | XcalableMP of last year (/peak) |
| 1 | **43 (67.50%)** | 43 (67.49%) | 29 (46.15%) |
| 4 | **173 (67.52%)** | 173 (67.49%) | 118 (46.11%) |
| 16 | **691 (67.49%)** | 691 (67.49%) | 470 (46.06%) |
| 64 | **2,764 (67.49%)** | 2,764 (67.48%) | 1,881 (46.11%) |
| 256 | **11,057 (67.48%)** | 11,056 (67.48%) | 7,528 (46.10%) |
| 1,024 | **44,227 (67.49%)** | 44,227 (67.49%) | 30,113 (46.11%) |
| 4,096 | **176,908 (67.49%)** | 176,909 (67.49%) | 120,450 (46.11%) |
| 16,384 | **707,638 (67.49%)** | 707,634 (67.49%) | 481,808 (46.10%) |
| 65,536 | **2,830,648 (67.49%)** | 2,830,612 (67.49%) | 1,927,172 (45.95%) |
| 82,944 | **3,582,500 (67.49%)** | 3,582,519 (67.49%) | 2,439,053 (45.95%) |

## 3.4 RandomAccess in XcalableMP

### 3.4.1 Overview

We implemented a RandomAccess algorithm written in XMP C language. The SLOC is **253**.

### 3.4.2 Differences from the XMP implementation of last year

In last year, the **post** and **wait** directives were implemented using MPI_Send/Recv in the Omni compiler. In order to improve performance, we have implemented the **post** and **wait** directives using RDMA of the K computer. Besides, we have fixed a minimum bug for verification.

### 3.4.3 Implementation

The XMP RandomAccess is iterated over sets of CHUNK updates on each node. In each iteration, the algorithm calculates for each update the destination node that owns the array element to be updated and communicates the data with each node. This communication pattern is known as complete exchange or all-to-all personalized communication, which can be performed efficiently by an algorithm referred to as the recursive exchange algorithm when the number of nodes is a power of two [19]. We implemented an algorithm with a set of remote writes to a coarray in local-view programming using XMP C language. Note that the number of the remote writes is also sent as an additional first element of the data. A point-to-point synchronization is specified with the XMP **post** and **wait** directives in order to realize asynchronous behavior of the algorithm.

The right code shows a part of the XMP RandomAccess code. Line 1 and 2 declare arrays *recv[][]* and *send[][]* as coarrays. In line 19, the variable *nsend*, which is the number of transfer elements, is set to the first element of array *send[][]* to be used by the destination node to update its local table. In line 20, elements from *send[isend][0]* to *send[isend][nsend]* are put to those from *recv[j][0]* to *recv[j][nsend]* in the *ipartner+1* node. In line 21, the **sync memory** directive is used to ensure the remote definition of a coarray is complete. In line 22 and 28, the **post** and **wait** directives are used for point-to-point synchronization. The **post** directive sends a signal to the node *ipartner+1* to inform that the remote definition for it is completed. Each node waits at the **wait** directive until receiving the signal from the node *jpartner+1*.

The Omni compiler has been optimized for the K computer. In order to use high-speed one-sided communication on the K computer, the coarray syntax is translated into calling the extended RDMA interface provided by the K computer.

```
 1 u64Int recv[MAXLOGPROCS][RCHUNK+1]:[*];
 2 u64Int send[2][CHUNKBIG+1]:[*];
 3 ...
 4 for (j = 0; j < logNumProcs; j++) {
 5   nkeep = nsend = 0;
 6   isend = j % 2;
 7   ipartner = (1 << j) ^ MyProc;
 8   if (ipartner > MyProc) {
 9     sort_data(data, data, &send[isend][1], nkept, ...);
10     if (j > 0) {
11       jpartner = (1 << (j−1)) ^ MyProc;
12 #pragma xmp wait(p(jpartner+1))
13 #pragma xmp sync_memory
14       nrecv = recv[j−1][0];
15       sort_data(&recv[j−1][1], data, &send[isend][1], nrecv, ...);
16     }
17   }
18   else { ... }
19   send[isend][0] = nsend;
20   recv[j][0:nsend+1]:[ipartner+1] = send[isend][0:nsend+1];
21 #pragma xmp sync_memory
22 #pragma xmp post(p(ipartner+1), 0)
23   if (j == (logNumProcs − 1)) update_table(data, Table, nkeep, ...);
24   nkept = nkeep;
25 }
26 ...
27 jpartner = (1 << (logNumProcs−1)) ^ MyProc;
28 #pragma xmp wait(p(jpartner+1))
29 #pragma xmp sync_memory
30 nrecv = recv[logNumProcs−1][0];
31 update_table(&recv[logNumProcs−1][1], Table, nrecv, ...);
```

### 3.4.4 Performance

We performed the XMP RandomAccess, referred to as flat-MPI. The table size is equal to half of the system memory. **Table 8** shows the performance results of this year and last year. For comparison, we also evaluated the modified hpcc-1.4 RandomAccess, for which the functions for sorting and updating the table are specifically optimized for the K computer. The best performance of the XMP RandomAccess is **254.20 GUP/s** (Giga UPdates per Second) for 16,384 nodes. Table 8 shows that the performance of the RandomAccess of this year is much better than that of the XMP RandomAccess of last year, and is a little better than that of modified hpcc-1.4.

**Table 8** Performance of RandomAccess in XcalableMP

| #Nodes | Performance (GUP/s) | | |
|---|---|---|---|
| | **XcalableMP of this year** | XcalableMP of last year | modified hpcc-1.4 |
| 1 | **0.09** | 0.08 | 0.08 |
| 8 | **0.51** | 0.43 | 0.44 |
| 64 | **2.92** | 2.08 | 2.64 |
| 512 | **16.94** | 11.41 | 15.78 |
| 4,096 | **83.61** | 61.43 | 80.81 |
| 16,384 | **254.20** | 162.63 | 243.40 |

## 4. Implementation and Performance of benchmarks using XcalableACC on HA-PACS/TCA

We are sorry that our implementations and tunings of HPL and FFT are not sufficient because of machine trouble of HA-PACS. We will improve them until BoF at SC14.

## 4.1 High-performance Linpack in XcalableACC

### 4.1.1 Overview

We implemented an HPL algorithm written in XACC C language. The SLOC is **343**.

### 4.1.2 Implementation

The following code shows a part of the XACC HPL code. This code is based on the XMP HPL code described in Section 3.1. We use cuBLAS library [20] to perform the DGEMM calculation instead of BLAS library.

The DGEMM calculation needs three arrays, *A[][]*, *A_L[]*, and *A_U[]* in Section 3.1. In the following code, *devA*, *devA_L*, and *devA_U* are their device pointers. The arrays *A_L[]* and *A_U[]* can be saved on GPU. In line 1 and 4, OpenACC **data** directive transfers the two arrays to GPU. However, the array *A[][]*, a coefficient matrix, is too large to be saved there. Thus, the XACC HPL splits the array *A[][]*, and transfers it to GPU. Finally, in order to set device pointers in the function *cublusDgemm()*, OpenACC **host_data** directive is inserted before it.

```
1 #pragma acc data copyin(devA_L[0:local_len_y], devA_U[0:local_len_x])
2 for(int n=0;n<local_len_y;n+=NB){
3    // Packing data to devA[] from A[][]
4 #pragma acc data copy(devA[0:local_len_x])
5 {
6 #pragma acc host_data use_device (devA_L, devA_U, devA)
7    cublasDgemm('n','n', local_len_x, NB, NB, −1.0, devA_U, local_len_x, devA_L+n*NB, NB, 1.0, devA, local_len_x);
```

### 4.1.3 Performance

We performed the XACC HPL with four processes on a single node, and each process operates a single GPU. The size of the coefficient matrix *A[][]* is about 50% of the system memory. **Table 9** shows the performance and the theoretical peak performance of the system. The best performance is **7,102.36** GFlops for 32 nodes. The performance is insufficient. The reason is that time of transfer data between CPU and host memory dominates the total computation time.

**Table 9** Performance of HPL in XcalableACC

| #Nodes | #CPUs | #GPUs | Performance **GFlops (/peak)** |
|---|---|---|---|
| 1 | 1 | 1 | **91.14 (5.94%)** |
| 1 | 2 | 4 | **284.54 (5.00%)** |
| 2 | 4 | 8 | **520.52 (4.76%)** |
| 4 | 8 | 16 | **1,021.81 (4.77%)** |
| 8 | 16 | 32 | **1,870.62 (4.42%)** |
| 16 | 32 | 64 | **3,757.59 (4.46%)** |
| 32 | 64 | 128 | **7,102.36 (4.22%)** |

### 4.1.4 More improvement

It is effective to overlap the DGEMM calculation and data transfer between host memory and GPU memory in XACC HPL. We are implementing it now.

## 4.2 Fast Fourier Transform in XcalableACC

### 4.2.1 Overview

Unfortunately, the result cannot be shown on time because of a severe machine trouble of HA-PACS. We introduce here how it has been implemented and will be tuned hereafter.

### 4.2.2 Implementation

Basically, the three XMP files developed for the K computer shown in Section 3.2 can be applied also in HA-PACS. They describe process parallelism among intra- and inter-node processes. About inside each process, instead of using zfft1d that is a FFTE routine for multi-thread, we will use both zfft1d and cuzfft1d that is another FFTE routine for GPU. Or, cuzfft1d may be ported into OpenACC because it may be more suitable in the context of XMP and/or XACC. Since zfft1d and (modified) cuzfft1d have the same interface, the caller written in XMP can handle them in the same manner.

Our basic approach is as follows. We employ three or more processes for each CPU group, which contains one CPU and two GPUs connected with the CPU. For the inter-process parallelism, the same program code as the one in the K computer can be used basically. For the inner-process parallelism, we use both zfft1d for CPUs and (modified) cuzfft1d for GPUs. Because every CPU is connected with two GPUs, at least one zfft1d and two (modified) cuzfft1d can be expected to be executed asynchronously. In order to simplify inter-process parallelism, every zfft1d and (modified) cuzfft1d process should bear the same amount of data. Load balance between a CPU and two GPUs can be adjusted with the number of processes of zfft1d and (modified) cuzfft1d.

### 4.2.3 Performance

The performance result will be opened at BoF in SC14.

## 4.3 STREAM in XcalableACC

### 4.3.1 Overview

We implemented a STREAM algorithm written in XACC C language. The SLOC is **84**.

### 4.3.2 Implementation

The XACC STREAM uses both CPUs and GPUs together. The following code shows a part of the XACC STREAM code. This code is based on the XMP STREAM code described in Section 3.3. Line 3 sets memory size for GPU per process. Line 4 transfers local arrays *a[]*, *b[]*, and *c[]* needed by GPU to the GPU. In lines 8-10, the OpenACC **parallel** directive parallelizes the loop statement asynchronously on GPU. In lines 12-14, the OpenMP **parallel** directive parallelizes the rest statement on CPU. In line 16, the OpenACC **wait** directive guarantees the asynchronous operation of lines 8-10 is complete.

```
 1  #pragma xmp nodes p(*)
 2
 3  int GPU_SIZE = size * ratio;
 4  #pragma acc data copy(a[0:GPU_SIZE], b[0:GPU_SIZE], c[0:GPU_SIZE])
 5  {
 6  for(k=0; k<NTIMES; k++) {
 7    ..
 8  #pragma acc parallel loop async
 9      for (j=0; j<GPU_SIZE; j++)
10        a[j] = b[j] + scalar*c[j];
11
12  #pragma omp parallel for
13    for (j=GPU_SIZE; j<size; j++)
14      a[j] = b[j] + scalar*c[j];
15
16  #pragma acc wait
17    ..
18  } // for
19  } // acc data
20  ..
21  #pragma xmp reduction(+:triadGBs)
```

### 4.3.3 Performance

We performed the XACC STREAM with four processes on one node, and each process has five threads. The vector lengths of the arrays *a[]*, *b[]*, and *c[]* are 357,913,942 which occupies 25% of the system memory. This vector length is a minimum value according to the HPCC specification [21] because we want to store data on GPU memory as much as possible. Each GPU memory can allocate 5.5GB by using OpenACC directive. While a single GPU memory stores 5.5GB data, a single process stores 2.5GB data on host memory. Therefore, we set the local variable *ratio* in line 3 of above code to 0.68. Note that this benchmark using the variable *ratio* is limited by CPU memory band width. The reason is that the CPU job, size of which is 2.5GB, will not finish until the GPU job, size of which is 5.5GB, will be completed. While the theoretical performance of the computation node is 1119.4 GB/s (= 250 GB/s × 4 + 59.7 GB/s × 2 CPUs), the real theoretical performance on this condition is 382.08 GB/s (= 59.7 GB/s × 2 CPUs ÷ (2.5 GB ÷ 8.0 GB)).

**Table 10** shows the performances where the "peak" value is 1119.4 GB/s. For comparison, we also evaluated the XMP STREAM benchmark which uses only CPU. The best performance of the XACC STREAM is **7,238.10 GB/s** for 32 nodes. The performance of the XACC STREAM is much better than that of the XMP STREAM. If the "peak" value in Table 10 is 382.08 GB/s, the XACC performance is about 60 % of the peak performance (e.g. 226.96 ÷ 382.08 = 0.59). Moreover, the XMP performance is also about 60 % of it if the "peak" value in Table 10 is 59.7 GB/s (e.g. 72.74 ÷ (59.7 × 2) = 0.61). Thus, we consider that the performances of XACC STREAM is reasonable.

**Table 10** Performance of STREAM in XcalableACC

| #Nodes | #CPUs | #GPUs | Performance (GB/s) | |
|---|---|---|---|---|
| | | | **XcalableACC (/peak)** | XcalableMP (/peak) |
| 1 | 1 | 1 | **112.98 (20.19%)** | 36.16 (6.46%) |
| 1 | 2 | 4 | **226.96 (20.28%)** | 72.74 (6.50%) |
| 2 | 4 | 8 | **453.71 (20.27%)** | 145.18 (6.48%) |
| 4 | 8 | 16 | **906.61 (20.25%)** | 289.99 (6.48%) |
| 8 | 16 | 32 | **1,812.30 (20.24%)** | 580.23 (6.48%) |
| 16 | 32 | 64 | **3,623.14 (20.23%)** | 1,159.52 (6.47%) |
| 32 | 64 | 128 | **7,238.10 (20.21%)** | 2,318.59 (6.47%) |

## 4.4 Himeno Benchmark in XcalableACC

### 4.4.1 Overview

We implemented a HIMENO algorithm written in XACC C language. The SLOC is **213**.

### 4.4.2 Implementation

In order to evaluate the productivity and the performance of the XACC programming model, we use the HIMENO Benchmark [8] that evaluates the performance of incompressible fluid analysis code in Flops. The reason the HIMENO Benchmark was selected is because it provides a good example of a stencil application benchmark and can be used to demonstrate parallelization by XACC **shadow** and **reflect** directives.

The right code shows a part of the XACC Himeno Benchmark code. In lines 1 to 5, the distributed arrays are defined. In line 6, the **shadow** directive defines a halo region of the first and second dimension in the distributed array *p[][][]*. The **reflect_init** of line 10 and the **reflect_do** of line 34 directives perform initialization and synchronization the halo region of the array *p[][][]* on accelerator memory. The two loop statements in lines 13-23 and 25-32 are distributed on each node, and thread-parallelization is performed on the accelerator.

```
 1  #pragma xmp template t(0:MKMAX, 0:MJMAX, 0:MIMAX)
 2  #pragma xmp nodes n(NDY, NDX)
 3  #pragma xmp distribute t(∗, block, block) onto n
 4  static float p[MIMAX][MJMAX][MKMAX];
 5  #pragma xmp align [k][j][i] with t(i, j, k) :: p, ..
 6  #pragma xmp shadow p[1:1][1:1][0]
 7  ...
 8  #pragma acc data copy(p, ... )
 9  {
10  #pragma xmp reflect_init (p) acc
11  for(n=0 ; n<nn ; ++n){
12    ...
13  #pragma xmp loop (k,j,i) on t(k,j,i)
14  #pragma acc parallel loop collapse(2) reduction(+:gosa) ...
15    for(i=1 ; i<imax−1 ; ++i)
16      for(j=1 ; j<jmax−1 ; ++j){
17  #pragma acc loop vector reduction(+:gosa) private(s0, ss)
18        for(k=1 ; k<kmax−1 ; ++k){
19          s0 = p[i+1][j][k] ∗ ...
20          ss = ...
21          gosa += ss∗ss;
22        }
23      }
24
25  #pragma xmp loop (k,j,i) on t(k,j,i)
26  #pragma acc parallel loop collapse(2) ...
27      for(i=1 ; i<imax−1 ; ++i)
28        for(j=1 ; j<jmax−1 ; ++j){
29  #pragma acc loop vector
30          for(k=1 ; k<kmax−1 ; ++k)
31            p[i][j][k] = wrk2[i][j][k];
32        }
33  ...
34  #pragma xmp reflect_do (p) acc
35  ...
36  } /∗ end n loop ∗/
37  } /∗ end of acc data ∗/
```

While the SLOC of the XACC Himeno Benchmark is **213**, that of the MPI Himeno Benchmark [8] is **325**. For comparison purposes, we have also implemented OpenACC and MPI Himeno Benchmark (OpenACC+MPI Himeno Benchmark) based on the MPI Himeno Benchmark. The MPI Himeno Benchmark synchronizes the halo region using MPI_Isend()/Irecv(). In the OpenACC+MPI Himeno Benchmark, we used GPUDirect RDMA to transfer the halo region on accelerator memory by adding the OpenACC **host_data** directives with the **use_device** clause before MPI_Isend()/Irecv(). In addition, we used the OpenACC **data** and **loop** directives to parallelize loop statement as same as the XACC Himeno Benchmark. The SLOC of the OpenACC+MPI Himeno Benchmark is **365**. The OpenACC+MPI Himeno Benchmark requires numerous lines to calculate the start and end indexes for loop statement on each process, and to transfer the halo region. By contrast, the XACC Himeno Benchmark does not need to calculate these indexes, and to only add XACC directives to transfer the halo region. From these implementations, we consider that XACC has a better productivity than the combination of OpenACC and MPI.

### 4.4.3 Performance

We set the number of elements of the array *p[256 × n][256][512]*, *n* is a number of processes in line 4 of the above code. We performed the XACC Himeno Benchmark with four processes on a single node. For comparison, we also evaluated the OpenACC+MPI Himeno Benchmark and the MPI Himeno Benchmark. We performed the MPI Himeno Benchmark, referred to as flat-MPI, which used only CPUs. **Table 11** shows the performances where the best performance of the XACC Himeno Benchmark is **6,871 GFlops** for 32 nodes. The result in Table 11 indicates that the performance of XACC is almost the same as that of the OpenACC + MPI Himeno Benchmark, and is about eight times better than that of the MPI Himeno Benchmark. The eight times is the same number as the difference between the bandwidth of GPU memory (1000 GB/s) and that of host memory (119.4 GB/s). Thus, we consider that the performances of XACC HIMENO benchmark is reasonable.

**Table 11** The performance of the Himeno Benchmark in XcalableACC

| #Nodes | #CPUs | #GPUs | Performance (GFlops) | | |
|---|---|---|---|---|---|
| | | | **XcalableACC (/peak)** | OpenACC+MPI (/peak) | MPI (/peak) |
| 1 | 1 | 1 | **56.27 (1.35%)** | 56.81 (1.38%) | 13.81 (0.17%) |
| 1 | 2 | 4 | **218.73 (1.35%)** | 222.72 (1.37%) | 27.04 (0.17%) |
| 2 | 4 | 8 | **437.15 (1.35%)** | 444.20 (1.37%) | 53.29 (0.16%) |
| 4 | 8 | 16 | **868.74 (1.34%)** | 886.95 (1.36%) | 105.72 (0.16%) |
| 8 | 16 | 32 | **1,734.18 (1.33%)** | 1,768.40 (1.36%) | 208.80 (0.16%) |
| 16 | 32 | 64 | **3,466.15 (1.33%)** | 3,515.03 (1.35%) | 414.18 (0.16%) |
| 32 | 64 | 128 | **6,870.98 (1.32%)** | 6,897.32 (1.33%) | 820.22 (0.16%) |

## 5. Conclusion

This report has investigated the productivity and the performance of the XMP and the XACC programming models through the HPCC Benchmarks and the Himeno Benchmark. The XMP programming model has a rich set of features based on global-view and local-view memory models that allows programmers to develop parallel applications with a little cost. Moreover, the XACC programming model is a directive-based language extension for an accelerator cluster, with which a programmer can develop applications via XMP and OpenACC directives easily.

## Acknowledgment

- The present study was supported by the JST/CREST 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.”
- Usage of HA-PACS/TCA system was supported by the “Interdisciplinary Computational Science Program” in the Center for Computational Sciences, University of Tsukuba.
- The modification of the two functions for sorting and updating local table in RandomAccess were done in cooperation with Fujitsu Limited.

### References

[1] Hitoshi Murai and Mitsuhisa Sato. “An Efficient Implementation of Stencil Communication for the XcalableMP PGAS Parallel Programming Language,” 7th International Conference on PGAS Programming Models, Edinburgh, Scotland, UK, October, 2013.

[2] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, and Mitsuhisa Sato. “Productivity and Performance of the HPC Challenge Benchmarks with the XcalableMP PGAS language,” 7th International Conference on PGAS Programming Models, Edinburgh, Scotland, UK, October, 2013.

[3] Masahiro Nakao, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato. “Productivity and Performance of Global-View Programming with XcalableMP PGAS Language,” CCGrid 2012 - The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Ottawa, Canada, May, 2012.

[4] Jinpil Lee. “A Study on Productive and Reliable Programming Environment for Distributed Memory System,” March, 2012.

[5] `http://www.aics.riken.jp/en/k-computer/about/`

[6] Hitoshi Murai, Masahiro Nakao, Takehiro Shimosaka, Akihiro Tabuchi, Taisuke Boku, and Mitsuhisa Sato. “XcalableACC - a Directive-based Language Extension for Accelerated Parallel Computing,” SC14 poster, New Orleans, LA, USA, Nov. 2014. (to be published)

[7] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Akihiro Tabuchi, Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, Mitsuhisa Sato. “XcalableACC: Extension of XcalableMP PGAS Language using OpenACC for Accelerator Clusters,” 1st Workshop on accelerator programming using directives (WACCPD), New Orleans, LA, USA, Nov. 2014. (to be published)

[8] The Riken Himeno CFD Benchmark. `http://accc.riken.jp/2444.htm`

[9] “HA-PACS Project”, `http://www.ccs.tsukuba.ac.jp/CCS/eng/research-activities/projects/ha-pacs`

[10] `http://www.pccluster.org/en/`

[11] C.H. Koelbel, D.B. Loverman, R. Shreiber, GL. Steele Jr., and M.E. Zosel. “The High Performance Fortran Handbook,” MIT Press, 1994.

[12] Ken Kennedy, Charles Koelbel, and Hans Zima. “The rise and fall of High Performance Fortran: an historical object lesson,” Proceedings of the third ACM SIGPLAN conference on History of programming languages, Pages 7-1-7-22, 2007

[13] R. Numwich and J. Reid. “Co-Array Fortran for parallel programming,” Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[14] `http://www.openacc-standard.org`

[15] Omni XcalableMP Compiler. `http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/xcalablemp/`

[16] `http://www.aics.riken.jp/jp/outreach/photogallery.html`

[17] `http://www.ffte.jp`

[18] Masahiro Nakao, Hitoshi Murai, Takenori Shimosaka, Mitsuhisa Sato. “HPC Challenge Award Competition Class 2,” `http://xcalablemp.org/download/publication/2013/HPCC13_XMP.pdf`

[19] R. Ponnusamy, A. Choudhary and G. Fox. “Communication Overhead on CM5: An Experimental Performance Evaluation,” Proc. Frontiers ’92, pp.108–115, 1992.

[20] cuBLAS, `https://developer.nvidia.com/cublas`

[21] HPC Challenge Website. `http://icl.cs.utk.edu/hpcc/software/index.html`