Implementation and Performance Evaluation of Omni Compiler

Masahiro Nakao and Hitoshi Murai

Abstract This chapter describes the implementation and performance evaluation of Omni compiler, which is a reference implementation of the compiler for XcalableMP. For performance evaluation, this chapter also presents how to implement the HPC Challenge benchmarks, which is a benchmark suite for an HPC parallel language. The results show that the performance of XMP is comparable to that of MPI in many cases.

1 Overview

Omni compiler is a source-to-source compiler that translates a sequential code in C and Fortran with XcalableMP (XMP), XcalableACC (XACC), and OpenACC directives into a parallel code (<https://omni-compiler.org>). The translated parallel code is compiled with a native compiler linked with Omni compiler runtime library. Omni compiler has been developed by Programming Environment Research Team of RIKEN Center for Computational Science [1] and HPCS laboratory [2] of University of Tsukuba in Japan.

2 Implementation

2.1 Operation Flow

In Omni compiler, XcodeML [3] is used to analyze a code in an intermediate code format of XML expression. Figure 1 shows an operation flow of Omni compiler. Firstly, Omni compiler translates directives in a user code into the runtime functions.

M. Nakao (✉) · H. Murai
RIKEN Center for Computational Science, Kobe, Hyogo, Japan
e-mail: masahiro.nakao@riken.jp; h-murai@riken.jp

Fig. 1 Operation flow of Omni compiler (<https://omni-compiler.org>)

If necessary, a code besides the directives is also modified. Secondly, a native compiler (e.g., gcc or Intel) compiles the translated code and creates an execution binary with linking to Omni compiler runtime library. The runtime library uses MPI in XMP, and CUDA in OpenACC, and both MPI and CUDA in XACC. As for XMP, Omni compiler may create better runtime libraries by adding a one-sided communication library to MPI, which is described in Chap. 3.

2.2 Example of Code Translation

This section describes how Omni compiler translates a user code for the global-view memory model. A code translation for the local-view memory model is described in Chap. 3.

2.2.1 Distributed Array

Figure 2 shows an XMP example code using an **align** directive to declare a distributed array `a[i][j]`.

Fig. 2 Code translation of align directive

Firstly, Omni compiler deletes a declaration of a local array $a[[]]$ and the **align** directive. Next, Omni compiler creates a descriptor `_XMP_DESC_a` by a function `_XMP_init_array_desc()` to set information of the distributed array. Omni compiler also adds a function `_XMP_alloc_array()` to allocate memory for the distributed array, and it sets values in an address `_XMP_ADDR_a` and a leading dimension `_XMP_ACC_a_0`. Note that a multidimensional distributed array is expressed as a one-dimensional array in the translated code since the size of each dimension of the array may be determined dynamically.

2.2.2 Loop Statement

Figure 3 shows an XMP example code using a **loop** directive to parallelize the following nested loop statement depending on the template t . Each dimension of t is distributed onto two nodes, which is omitted there.

In the translated code above, a pointer `_XMP_MULTI_ADDR_a` is used which has the size of each dimension as a head pointer of the distributed array $a[[]]$. To improve performance, operations in a loop statement are performed using the pointer [4]. Note that this pointer can be used when the number of elements in each dimension of a distributed array is divisible by the number of nodes. If the condition is not met, a one dimensional pointer `_XMP_ADDR_a` and an offset `_XMP_ACC_a_0` are used as shown in the translated code below.

Moreover, because values in ending conditions of the loop statement ($i < 10, j < 10$) are constants in a pre-translated code and are divisible by the number of nodes,

```
#pragma xmp loop on t[i][j]
for(int i=0;i<10;i++)
  for(int j=0;j<10;j++)
    a[i][j] = ...
```

```
double (*_XMP_MULTI_ADDR_a)[5] = (double (*)(5))(_XMP_ADDR_a);
for(int i=0;i<5;i++)
  for(int j=0;j<5;j++)
    _XMP_MULTI_ADDR_a[i][j] = ...
```

or

```
for(int i=0;i<5;i++)
  for(int j=0;j<5;j++)
    *(_XMP_ADDR_a + i * _XMP_ACC_a_0 + j) = ...
```

Fig. 3 Code translation of loop directive

```
int b[10];
:
#pragma xmp bcast (b)
```

```
_XMP_M_BCAST_EXEC(b, 10, sizeof(int));
```

Fig. 4 Code translation of bcast directive

the values are translated to constants ($i < 5, j < 5$) automatically. If the values are variables in the pre-translated code or not divisible by the number of nodes, the runtime function is inserted just before the loop statement to calculate values for ending conditions. The calculated values are set in newly created variables.

2.2.3 Communication

Figure 4 shows an XMP example code using a **bcast** directive to broadcast a local array b . Basically translations of communication directives are simple. The runtime functions call MPI functions directly.

3 Installation

This section describes how to install the latest Omni compiler version 1.3.2. Omni compiler is installed by a general installation method on UNIX (`./configure; make; make install`). When executing `./configure` without options, only XMP is installed. When installing OpenACC and/or XACC, it is required for some options to “`./configure`”, which is described in Sect. 3.5.

3.1 Overview

We provide two versions of Omni compiler, the one is “stable version” and the other is “nightly build version.” While the stable version is a so-called official version that has a version number, the nightly build version is a trial version that is released at midnight on our website (<https://omni-compiler.org>). Omni compiler is developed in GitHub repository (<https://github.com/omni-compiler/omni-compiler>). Our web server gets the source code from the GitHub repository and generates the nightly build version every day.

3.2 *Get Source Code*

3.2.1 From GitHub

Please visit the GitHub repository (<https://github.com/omni-compiler/omni-compiler>) which provides only nightly build version. Otherwise, please execute the following `git` command.

```
$ git clone --recursive https://github.com/omni-compiler/omni-compiler.git
```

Note that the source code of Omni compiler does not contain that of XcodeML, so the `--recursive` option is required. As a supplement, XcodeML is also developed in the GitHub repository (<https://github.com/omni-compiler/xcodeml-tools>).

3.2.2 From Our Website

Please visit our website (<https://omni-compiler.org>) which provides packages of stable version and nightly build version. The package of nightly build version is generated every midnight around 12:00 a.m. (JST) if the latest GitHub repository was updated yesterday. These packages contain XcodeML.

3.3 *Software Dependency*

Before installation of Omni compiler, the following software must be installed.

```
yacc, lex, C Compiler (C99 or over), Fortran Compiler (Fortran 2008 or over),  
Java Compiler, MPI (version 2 or over), libxml2, make
```

3.4 *General Installation*

This section explains how to install Omni compiler in a general Unix environment.

3.4.1 Build and Install

```
$ ./configure --prefix=(INSTALL PATH)
$ make
$ make install
```

3.4.2 Set PATH

- bash and zsh

```
$ export PATH=(INSTALL PATH)/bin:$PATH
```

- csh and tcsh

```
% setenv PATH (INSTALL PATH)/bin:$PATH
```

3.5 *Optional Installation*

3.5.1 OpenACC

Please add “--enable-openacc” and “--with-cuda=(CUDA PATH)” options to “./configure”.

```
$ ./configure --enable-openacc --with-cuda=(CUDA PATH)
$ make
$ make install
```

It may be possible to generate a more suitable runtime library by adding options to “nvcc” command, which is used to generate the runtime library for OpenACC and XACC. In that case, please also add the “--with-gpu-cflags=(NVCC CFLAGS)” option.

```
$ ./configure --enable-openacc --with-cuda=(CUDA PATH) --with-gpu-  
cflags=(NVCC CFLAGS)  
$ make  
$ make install
```

3.5.2 XscalableACC

Please add “--enable-openacc --enable-xacc” to “./configure”.

```
$ ./configure --enable-openacc --enable-xacc  
$ make  
$ make install
```

As with OpenACC, if necessary, please add the “--with-cuda=(CUDA PATH)” and “--with-gpu-cflags=(NVCC CFLAGS)” options to “./configure”.

3.5.3 One-Sided Library

Omni compiler may generate a better runtime library by a one-sided library for XMP. Omni compiler supports the following one-sided libraries.

- Fujitsu MPI Extended RDMA (FJRDMA)
It is low-level communication layer for Fujitsu machines (e.g., the K computer, FX100, and FX10). When using it, please specify a target machine to ./configure. (e.g., “\$./configure --target=FX100-linux-gnu”)
- GASNet (<https://gasnet.lbl.gov>)
It is a one-sided communication library developed by U.C. Berkeley. When using it, please specify “install path of GASNet” and “its conduit” to ./configure. (e.g., \$./configure --with-gasnet=/usr --with-gasnet-conduit=ibv)
- MPI version 3
Omni compiler automatically selects MPI version 3 under the following conditions.
 - MPI implementation supports MPI version 3
 - Specifying neither FJRDMA nor GASNet.

4 Creation of Execution Binary

This section describes how to create an execution binary from a code with XMP, XACC, and OpenACC directives, and how to execute it. Note that Omni compiler supports only C language for OpenACC.

4.1 *Compile*

- XMP in C language

```
$ xmpcc a.c
```

- XMP in Fortran

```
$ xmpf90 a.f90
```

- XACC in C language

```
$ xmpcc -xacc a.c
```

- XACC in Fortran

```
$ xmpf90 -xacc a.f90
```

- OpenACC in C language

```
$ ompcc -acc a.c
```


A native compiler finally compiles the code translated by Omni compiler. Thus, all compile options of XMP are passed to the native compiler. For example, when using the optimization option “-O2”, it is passed to the native compiler.

```
$ xmpcc -O2 a.c
```

4.2 Execution

4.2.1 XcalableMP and XcalableACC

Because the runtime libraries of XMP and XACC use MPI, a program is executed via an MPI execution command (e.g., “`mpiexec`”). However, when using GASNet, a program is executed via a GASNet execution command (e.g., “`gasnetrun_ibv`”).

```
$ mpiexec -n 2 ./a.out
```

```
$ gasnetrun_ibv -n 2 ./a.out
```

4.2.2 OpenACC

```
$ ./a.out
```

4.3 Cooperation with Profiler

In order to improve the performance of an application, it is useful to take a profile. Omni compiler has a function to cooperate with Scalasca (<https://www.scalasca.org>) and tlog which are profiling tools. The function can profile the execution of XMP directives. Note that the function supports only the XMP in C language now.

4.3.1 Scalasca

Scalasca is an opensource software that measures and analyzes the runtime behaviors.

When profiling all XMP directives that exist in code, please add the “`--profile scalasca`” option to a compile command.

```
$ xmpcc --profile scalasca a.c
```

When profiling selected XMP directives there, please add the “`profile`” clause to the directives and the “`--selective-profile scalasca`” option to a compile command.

```
#pragma xmp bcast (a) profile
```

```
$ xmpcc --selective-profile scalasca a.c
```

Figure 5 shows an example of profiling by Scalasca.

Fig. 5 Profile by Scalasca (<https://omni-compiler.org>)

4.3.2 tlog

Omni compiler package contains tlog that measures executing time of the XMP directives.

When profiling all XMP directives that exist in code, please add the “`--profile tlog`” option to a compile command.

```
$ xmpcc --profile tlog a.c
```

When profiling selected XMP directives there, please add the “`profile`” clause to the directives as in Sect. 4.3.1 and the “`--selective-profile tlog`” option to a compile command.

```
$ xmpcc --selective-profile tlog a.c
```

After executing a program, tlog generates a file “`trace.log`” which stores profiling results. To open the result, please use the “`tlogview`” command. Figure 6 shows an example of profiling by tlog.

```
$ tlogview trace.log
```

Fig. 6 Profile by tlog (<https://omni-compiler.org>)

5 Performance Evaluation

In order to evaluate the performance of XMP, we implemented the HPC Challenge (HPCC) benchmark (<https://icl.utk.edu/hpcc/>), namely, EP STREAM Triad (STREAM), High-Performance Linpack (HPL), Global fast Fourier transform (FFT), and RandomAccess [5]. While the HPCC benchmark is used to evaluate multiple attributes of HPC systems, the benchmark is also useful to evaluate the properties of a parallel language. The HPCC benchmark was used at the HPCC Award Competition (<https://www.hpccchallenge.org>). The HPCC Award Competition consists of two classes. While the purpose of class 1 is to evaluate the performance of a machine, the purpose of class 2 is to evaluate both the productivity and performance of a parallel programming language. XMP won the class 2 prizes in 2013 and 2014.

5.1 Experimental Environment

For performance evaluation, this section uses 16,384 compute nodes on the K computer and 128 compute nodes on a Cray CS300 system named “the COMA system.” Tables 1 and 2 show the hardware specifications and software environments.

For comparison purposes, this section also evaluates the HPCC benchmark in C language and MPI library. We execute STREAM, HPL, and FFT with eight threads per process on each CPU of the K computer, and with ten threads per process on each CPU of the COMA system. Since RandomAccess is not parallelized with threads and can be executed by the power of only two processes, we execute it with eight processes on each CPU of both systems.

The specification of HPCC Award Competition class 2 defines the minimum problem size for each benchmark. While the main array of HPL should occupy at least half of the system memory, the main arrays of STREAM, FFT, and RandomAccess should occupy at least a quarter of the system memory. We set each

Table 1 Experimental environment for the K computer

CPU	SPARC64 VIIIfx 2.0 GHz, 8 Cores
Memory	DDR3 SDRAM 16 GB, 64 GB/s
Network	Torus fusion six-dimensional mesh/torus network, 5 GB/s $\times$ 10
Library	Fujitsu Compiler K-1.2.0-19, Fujitsu MPI K-1.2.0-19, Fujitsu SSLII K-1.2.0-19

Table 2 Experimental environment for the COMA system

CPU	Xeon E5-2670v2, 2.5 GHz (Turbo Boost 3.3 GHz), 10 Cores $\times$ 2CPUs
Memory	DDR3 SDRAM 64 GB, 119.4 GB/s (= 59.7 GB/s $\times$ 2 CPUs)
Network	InfiniBand FDR, fat-tree, 7 GB/s
Library	Intel Compiler 15.0.5, Intel MPI 5.1.1, GASNet 1.26.0, Intel MKL 11.2.4

problem size to be equal to the minimum size. As for coarray syntax, Omni compiler uses FJRDMA on the K computer and uses GASNet on the COMA system.

5.2 EP STREAM Triad

5.2.1 Design

STREAM measures the memory bandwidth to use simple vector kernel ($a \leftarrow b + \alpha c$). STREAM is so straightforward that its kernel does not require communication.

5.2.2 Implementation

Figure 7 shows a part of the STREAM code. In line 1, the **node** directive declares a node array p to parallelize the program. In line 2, normal arrays $a[]$, $b[]$, and $c[]$, and a scalar value *scalar* are declared. In lines 5 and 14, the **barrier** directive is inserted before `xmp_wtime()` to measure time. The directives of lines 8–9 are optimization directives for the Fujitsu compiler. While the **#pragma loop xfill** ensures one cache line to store write-only data, the **#pragma loop noalias** indicates that different pointer variables cannot possibly indicate the same storage area. These optimization directives are used on only the K computer. In lines 10–12, STREAM kernel is parallelized by the OpenMP **parallel** directive. In line 17, **local_performance()**

```

1  #pragma xmp nodes p[*]
2  double a[N], b[N], c[N], scalar;
3  ...
4  for(k=0;k<TIMES;k++){
5  #pragma xmp barrier
6      times[k] = -xmp_wtime();
7
8  #pragma loop xfill
9  #pragma loop noalias
10 #pragma omp parallel for
11     for (i=0; i<N; i++)
12         a[i] = b[i] + scalar * c[i];
13
14 #pragma xmp barrier
15     times[k] += xmp_wtime();
16 }
17 double performance = local_performance(time, TIMES, N);
18 #pragma xmp reduction(+:performance)

```

Fig. 7 Part of the STREAM code [5]

calculates the performance on each node locally. In line 18, the **reduction** directive performs a reduction operation among nodes to calculate the total performance.

5.2.3 Evaluation

First of all, in order to consider the effectiveness of **#pragma loop xfill** and **#pragma loop noalias**, we evaluate STREAM with and without these directives on a single node of the K computer. We also insert these directives into the MPI implementation for evaluation. Figure 8 shows that the performance results with these directives are about 1.46 times better than those without the directives. Therefore, we use the directives in next evaluations.

Figure 9 shows the performance results and a comparative performance evaluation of both implementations. The comparative performance evaluation is called the “performance ratio.” When the performance ratio is greater than 1, the performance result of the XMP implementation is better than that of the MPI implementation. XMP’s best performance results are 706.38 TB/s for 16,384 compute nodes on the

Fig. 8 Preliminary evaluation of STREAM [5]

Fig. 9 Performance results for STREAM [5]

K computer, and 11.55 TB/s for 128 compute nodes on the COMA system. The values of the performance ratio are between 0.99 and 1.00 on both systems.

5.3 High-Performance Linpack

5.3.1 Design

HPL evaluates the floating point rate of execution for solving a linear system of equations. The performance result has been used in the TOP500 list (<https://www.top500.org>). To achieve a good load balance on HPL, we distribute the main array in a *block-cyclic* manner. Moreover, in order to achieve high performance with portability, our implementation calls BLAS [6] to perform the matrix operations. These techniques are inherited from the MPI implementation.

5.3.2 Implementation

Figure 10 shows that each dimension of the coefficient matrix $A[[]]$ is distributed in the *block-cyclic* manner. The **template** and the **nodes** directives declare a two-dimensional template t and node array p . The **distribute** directive distributes t onto $Q \times P$ nodes with the same block size NB . The **align** directive aligns $A[[]]$ with t .

HPL has an operation in which a part of the coefficient matrix is broadcast to the other process columns asynchronously. This operation, called “panel broadcast,” is one of the most important operations for overlapping panel factorizations and data transfer. Figure 11 shows the implementation that uses the **gmove** directive with the **async** clause. The second dimension of array $L[[]]$ is also distributed in a *block-cyclic* manner and $L[[]]$ is replicated. Thus, the **gmove** directive broadcasts elements $A[j:NB][j+NB:len]$ to $L[0:NB][j+NB:len]$ asynchronously.

Figure 12 shows that **cblas_dgemm()**, which is a BLAS function for a matrix multiplication, applies the distributed arrays $L[[]]$ and $A[[]]$. Note that **cblas_dgemm()** is executed by multiple threads locally. In lines 2–3, **xmp_desc_of()** gets descriptors of $L[[]]$ and $A[[]]$, and **xmp_array_lda()** gets the leading dimensions L_ld and A_ld . In line 5, the L_ld and A_ld are used in

Fig. 10 Block-cyclic distribution in HPL [5]

```

1 double L[NB][N];
2 #pragma xmp align L[*][i] with t[*][i]
3 ...
4 int len = N - j - NB;
5 #pragma xmp gmove async (tag)
6 L[0:NB][j+NB:len] = A[j:NB][j+NB:len];

```

Fig. 11 Panel broadcast in HPL [5]

```

1 int L_ld, A_ld;
2 xmp_array_lda(xmp_desc_of(L), &L_ld);
3 xmp_array_lda(xmp_desc_of(A), &A_ld);
4 ...
5 cblas_dgemm(..., &L[0][j], L_ld, ..., &A[i][j], A_ld, ...);

```

Fig. 12 Calling the function `cblas_dgemm()` in HPL [5]

The K computer

The COMA system

Fig. 13 Performance results for HPL [5]

`cblas_dgemm()`. Note that `L_ld` and `A_ld` remain unchanged from the beginning of the program, and so each `xmp_array_lda()` is called only once.

5.3.3 Evaluation

Figure 13 shows the performance results and performance ratios. XMP's best performance results are 402.01 TFlops (76.68% of the peak performance) for

4096 compute nodes on the K computer, and 47.32 TFlops (70.02% of the peak performance) for 128 compute nodes on the COMA system. The values of the performance ratio are between 0.95 and 1.09 on the K computer, and between 0.99 and 1.06 on the COMA system.

5.4 Global Fast Fourier Transform

5.4.1 Design

FFT evaluates the performance for a double-precision complex one-dimensional discrete Fourier transform. We implement a six-step FFT algorithm [7, 8] using FFTE library [9]. The six-step FFT algorithm is also used in the MPI implementation. In the six-step FFT algorithm, both the computing performance and the all-to-all communication performance for a matrix transpose are important. The six-step FFT algorithm reduces the cache-miss ratio by expression of a two-dimensional array. In order to develop the XMP implementation, we use XMP in Fortran because FFTE library is written in Fortran and therefore it is easy to call it. In addition, we use the XMP intrinsic subroutine `xmp_transpose()` to transpose a distributed array in the global-view memory model. Figure 14 shows an example of `xmp_transpose()`. The first argument is an output array, and the second argument is the input array. The third argument is an option to save memory, and is “0” or “1.” If it is “0,” an input array must not be changed. If it is “1,” an input

```

1 complex*16 a(4,12), b(12,4)
2 !$xmp template ty(12)
3 !$xmp template tx(4)
4 !$xmp nodes p(4)
5 !$xmp distribute ty(block) onto p
6 !$xmp distribute tx(block) onto p
7 !$xmp align a(*,i) with ty(i)
8 !$xmp align b(*,i) with tx(i)
9 call xmp_transpose(a, b, 1)

```

Fig. 14 Action of subroutine `xmp_transpose()` [5]

array may be changed but less memory may be used. Thus, we use “1” in the XMP implementation. In Fig. 14, the second dimensions of arrays $a()$ and $b()$ are distributed in the *block* manner, and array $b()$ is transposed to array $a()$. For example, elements $b(1:3,2)$ on $p(2)$ are transferred to elements $a(2,1:3)$ on $p(1)$.

5.4.2 Implementation

Figure 15 shows a part of the XMP implementation. In lines 1–9, arrays $a()$, $b()$, and $w()$ are distributed in a *block* manner. The $a()$ is aligned with template ty , and the $b()$ and $w()$ arrays are aligned with template tx . In lines 16–20, each thread on all nodes calls the FFTE subroutine $zfft1d()$, which applies the distributed array $b()$. Note that the subroutine $zfft1d()$ executes with a single thread locally. In lines 22–23, the XMP **loop** directive and the OpenMP **parallel** directive parallelize the loop statement. In line 30, **xmp_transpose()** is used to transpose the distributed two-dimensional array.

```

1  complex*16 a(nx,ny), b(ny,nx), w(ny,nx)
2  !$xmp template ty(ny)
3  !$xmp template tx(nx)
4  !$xmp nodes p(*)
5  !$xmp distribute ty(block) onto p
6  !$xmp distribute tx(block) onto p
7  !$xmp align a(*,i) with ty(i)
8  !$xmp align b(*,i) with tx(i)
9  !$xmp align w(*,i) with tx(i)
10
11 integer, save :: ithread
12 !$omp threadprivate (ithread)
13 !$omp parallel
14     ithread = omp_get_thread_num()
15 !$omp end parallel
16 !$xmp loop on tx(i)
17 !$omp parallel do
18     do i=1,nx
19         call zfft1d(b(1,i),ny,-1,cy(1,ithread))
20     end do
21
22 !$xmp loop on tx(i)
23 !$omp parallel do
24     do i=1,nx
25         do j=1,ny
26             b(j,i)=b(j,i)*w(j,i)
27         end do
28     end do
29
30 call xmp_transpose(a,b,1)

```

Fig. 15 Part of the FFT code [5]

5.4.3 Evaluation

Figure 16 shows the performance results and performance ratios. XMP’s best performance results are 39.01 TFlops for 16,384 compute nodes on the K computer, and 0.94 TFlops for 128 compute nodes on the COMA system. The values of the performance ratio are between 0.94 and 1.13 on the K computer, and between 0.94 and 1.12 on the COMA system.

5.5 RandomAccess

5.5.1 Design

RandomAccess evaluates the performance of random updates of a single table of 64-bit integers which may be distributed among processes. The random update for a distributed table requires an all-to-all communication. We implement a recursive exchange algorithm [10], as with the MPI implementation. The recursive exchange algorithm consists of multiple steps. A process sends a data chunk to another process in each step. Because RandomAccess requires a random communication pattern, as its name suggests, the pattern is not supported by the global-view memory model. Thus, we use the local-view memory model to implement RandomAccess. Note that the MPI implementation uses functions `MPI_Isend()` and `MPI_Irecv()`.

5.5.2 Implementation

A source node transfers a data chunk to a destination node, and then the destination node updates own table using the received data. The MPI implementation repeatedly executes the recursive exchange algorithm by 1024 elements in the table. The HPCC Award Competition class 2 specification defines the constant value 1024. The recursive exchange algorithm sends about half of the 1024 elements in each

Fig. 16 Performance results for FFT [5]

step. Therefore, the chunk size is about 4096 Bytes ($= 1024/2 \times 64 \text{ bits}/8$). Note that the destination node cannot know how many elements are sent by the source node. Thus, the MPI implementation gets the number of elements using the function **MPI_Get_count()**. We implement the algorithm using a coarray and the **post/wait** directives for the recursive exchange algorithm, and the number of elements is added to the first element of the coarray.

Figure 17 shows a part of the XMP implementation. In line 2, the coarrays *recv* and *send* are declared. In line 6, the data chunk size is set at the first element of the coarray, and it is put in line 7. In line 8, the node sends notification of the completion of the coarray operation of line 7 to the node *p[ipartner]*. In line 10, the node receives the notification from the node *p[jpartner]*, which ensures that the node *p[jpartner]* receives the data. In line 11, the node gets the number of elements in the received data. In line 12, the node updates own table by using the received data.

5.5.3 Evaluation

Figure 18 shows the performance results and performance ratios. The Giga-updates per second (GUPS) on the vertical axis is the measurement value, which is the number of update tables per second divided by 10^9 . XMP's best performance results are 259.73 GUPS for 16,384 compute nodes on the K computer, and 6.23 GUPS for 128 compute nodes on the COMA system. The values of the performance ratio are between 1.01 and 1.11 on the K computer, and between 0.57 and 1.03 on the COMA system. On the K computer, the performance results for the XMP implementation are always slightly better than those for the MPI implementation. However, on the COMA system, the performance results for the XMP implementation are worse than those for the MPI implementation using multiple CPUs.

```

1  #pragma xmp nodes p[*]
2  unsigned long long recv[ITER][LOGPROCS][CHUNK]:[*], send[2][CHUNKBIG]:[*];
3  ...
4  for(j=0;j<logNumProcs;j++){
5      ...
6      send[i][0] = nsend;
7      recv[iter_mod][j][0:nsend+1]:[ipartner] = send[i][0:nsend+1];
8      #pragma xmp post(p[ipartner], tag)
9      ...
10     #pragma xmp wait(p[jpartner], tag)
11     nrecv = recv[iter_mod][j-1][0];
12     update_table(&recv[iter_mod][j-1][1], ..., nrecv, ...);
13     ...
14 }

```

Fig. 17 Part of the RandomAccess code [5]

The K computer

The COMA system

Fig. 18 Performance results for RandomAccess [5]

5.6 Discussion

We implement STREAM, HPL, and FFT using the global-view memory model, which enables programmers to develop the parallel codes from the sequential codes using the XMP directives and functions easily. Specifically, in order to implement the parallel STREAM code, a programmer only adds the XMP directives into the sequential STREAM code. The XMP directives and existing directives, such as OpenMP directives and Fujitsu directives, can coexist. Moreover, existing high-performance libraries, such as BLAS and FFTE, can be used with an XMP distributed array. These features improve the portability and performance of XMP applications.

We also implement RandomAccess using the local-view memory model, where the coarray syntax enables a programmer to transfer data intuitively. In the evaluation, the performance of the XMP implementation is better than that of the MPI implementation on the K computer, but is worse than that of the MPI implementation on the COMA system.

To clarify the reason why XMP performance is dropped on the COMA system, we develop a simple ping-pong benchmark using the local-view memory model. The benchmark measures the latency for transferring data repeatedly between two nodes. For comparison purposes, we also implement one using **MPI_Isend()** and **MPI_Irecv()** that are used in the MPI version RandomAccess.

Figure 19 shows parts of the codes. In XMP of Fig. 19, in line 5, $p[0]$ puts a part of $src_buf[]$ into $dst_buf[]$ in $p[1]$. In line 6, the **post** directive ensures the completion of the coarray operation of line 5 and sends a notification to $p[1]$. In line 10, $p[1]$ waits until receiving the notification from $p[0]$. In line 11, $p[1]$ puts a part of $src_buf[]$ into $dst_buf[]$ in $p[0]$. In line 12, the **post** directive ensures the completion of the coarray operation of line 11 and sends a notification to $p[0]$. In line 7, $p[0]$ waits until receiving the notification from $p[1]$. Figure 19 also shows the ping-pong benchmark that uses MPI functions.

Figure 20 shows the latency for transferring data. The results on the K computer show that the latency for the XMP implementation is better than that for the

```

1  int me = xmpc_node_num();
2  int target = (me == 0)? 1 : 0;
3  ...
4  if(me == 0){
5      dst_buf[0:size]:[target] = src_buf[0:size];
6      #pragma xmp post(p[target], tag)
7      #pragma xmp wait(p[target], tag)
8  }
9  else if(me == 1){
10     #pragma xmp wait(p[target], tag)
11     dst_buf[0:size]:[target] = src_buf[0:size];
12     #pragma xmp post(p[target], tag)
13 }

```

```

1  int me;
2  MPI_Comm_rank(..., &me);
3  int target = (me == 0)? 1 : 0;
4  ...
5  if(me == 0){
6      MPI_Isend(src_buf, ..., target, ...);
7      MPI_Wait(...);
8      MPI_Irecv(dst_buf, ..., target, ...);
9      MPI_Wait(...);
10 }
11 else if(me == 1){
12     MPI_Irecv(dst_buf, ..., target, ...);
13     MPI_Wait(...);
14     MPI_Isend(src_buf, ..., target, ...);
15     MPI_Wait(...);
16 }

```

Fig. 19 Part of the ping-pong benchmark code in XMP and MPI [5]

The K computer

The COMA system

Fig. 20 Performance results for ping-pong benchmark [5]

MPI implementation for 2048 Bytes or greater transfer size on the K computer. In contrast, the results on the COMA system show that the latency of the XMP implementation is always worse than that of the MPI implementation on the COMA system. The latency of XMP with FJRDMA at 4096 Bytes, which is the average data chunk size, is $5.83\mu\text{s}$ and the latency of MPI is $6.89\mu\text{s}$ on the K computer. The latency of XMP with GASNet is $5.05\mu\text{s}$ and that of MPI is $3.37\mu\text{s}$ on the COMA system. Thus, we consider the reason for the performance difference of RandomAccess is the communication performance. The performance difference is also due to the differences in the synchronization mechanism of the one-sided XMP coarray and the two-sided MPI functions. Note that a real application would not synchronize after every one-sided communication. It is expected that a single synchronization should occur after multiple one-sided communications to achieve higher performance.

In addition, the performance results for HPL and FFT are slightly different from those for the MPI implementations. We consider that these differences are caused by small differences in the implementations. In HPL, for the panel-broadcast operation, the XMP implementation uses the **gmove** directive with the **async** clause, which calls **MPI_Ibcast()** internally. In contrast, the MPI implementation uses **MPI_Send()** and **MPI_Recv()** to perform the operation by the “modified increasing ring” [11]. In FFT, the XMP implementation uses XMP in Fortran, but the MPI implementation uses C language. Both implementations call the same FFTE library. In addition, the MPI implementation uses **MPI_Alltoall()** to transpose a matrix. Since **xmp_transpose()** calls **MPI_Alltoall()** internally, the performance levels for both **xmp_transpose()** and **MPI_Alltoall()** must be the same. Therefore, the language differences and refactoring may have caused the performance difference.

6 Conclusion

The chapter describes the implementation and performance evaluation of Omni compiler. We evaluate the performance of the HPCC benchmark in XMP on the K computer up to 16,384 compute nodes and a generic cluster system up to 128 compute nodes. The performance results for the XMP implementations are almost the same as those for the MPI implementations in many cases. Moreover, it demonstrates that the global-view and the local-view memory models are useful to develop the HPCC benchmark.

References

1. Programming Environment Research Team, <https://pro-env.riken.jp>
2. High Performance Computing System laboratory, University of Tsukuba, Japan, <https://www.hpcs.cs.tsukuba.ac.jp>

3. M. Sato et al., Omni compiler and XcodeML: an infrastructure for source-to-source transformation, in *Platform for Advanced Scientific Computing Conference (PASC16)*, Lausanne (2016)
4. M. Nakao et al., Performance evaluation for Omni XcalableMP compiler on many-core cluster system based on knights landing, in *IXPUG Workshop Asia 2018*, Tokyo (2018), pp. 52–58
5. M. Nakao et al., Implementation and evaluation of the HPC challenge benchmark in the XcalableMP PGAS language. *Int. J. High Perform. Comput. Appl.* **33**(1), 110–123 (2017)
6. BLAS: Basic Linear Algebra Subprograms, <http://www.netlib.org/blas/> (2016)
7. D.H. Bailey, FFTs in external or hierarchical memory. *J. Supercomput.* **4**, 23–35 (1990)
8. C. Van Loan, *Computational Frameworks for the Fast Fourier Transform* (Society for Industrial and Applied Mathematics, Philadelphia, 1992)
9. D. Takahashi, *A Fast Fourier Transform Package*, <http://www.ffte.jp> (2014)
10. R. Ponnusamy et al., Communication overhead on the CM5: an experimental performance evaluation, in *Fourth Symposium on the Frontiers of Massively Parallel Computation* (1992), pp. 108–115
11. HPL Algorithm Panel Broadcast, <http://www.netlib.org/benchmark/hpl/algorithm.html> (2016)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.