XcalableMP for Productivity and Performance in HPC Challenge Award Competition Class 2

Masahiro Nakao[1,a)] Hitoshi Murai[2] Takenori Shimosaka[2] Mitsuhisa Sato[1,2]

1. Center for Computational Sciences, University of Tsukuba, Japan
2. RIKEN Advanced Institute for Computational Science, Japan
a) mnakao@ccs.tsukuba.ac.jp

1. Summary

In this paper, we present XcalableMP implementations of the HPCC Benchmarks, namely, High-performance Linpack (HPL), RandomAccess, Fast Fourier Transform (FFT), and STREAM. Moreover, we have implemented the Himeno Benchmark [1] which is a typical stencil application.

The highlights of this submission are as follows:

- **Table 1** shows the SLOC (Source lines of code) of the implementations.
- **Table 2** shows an experimental environment. We used the K computer at RIKEN AICS in Japan.
- **Table 3** shows a performance summary.

Table 1 SLOC of the HPCC and the Himeno Benchmarks

	HPL	RandomAccess	FFT	STREAM	Himeno
XcalableMP	306	250	239†+ 283‡+ 1,892*	66	137
Reference	8,800	938	1,130	329	380**

† We have modified some files of the optimized hpcc-1.4 FFT [2]. In addition, we have implemented the main kernel of the FFTE.
‡ The XMP FFT directly uses some files of the optimized hpcc-1.4 FFT. These files do not change.
* The XMP FFT calls the some FFTE kernel by using C interface.
** The original Himeno Benchmark is written in Fortran90 + MPI.

Table 2 Experimental environment of 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 x 10
Compier	Fujitsu C/Fortran Compiler K-1.2.0-14
Comm. Library	Fujitsu MPI K-1.2.0-14
BLAS	Fujitsu SSLII K-1.2.0-14

Table 3 Performance summary of the HPCC and the Himeno Benchmarks

Benchmark	#Nodes	#Cores	Performance	of peak
HPL	8,192	65,536	542.94 TFlops	53.02%
RandomAccess	16,384	131,072	162.63 GUP/s	-
FFT	9,216	73,728	24.67 TFlops	2.14%
STREAM	8,192	65,536	331.55 TB/s	64.76%
Himeno	16,384	131,072	299.10 TFlops	14.60%

2. Overview of XcalableMP

XcalableMP [3–6], XMP for short, is a directive-based language extension for distributed memory systems, which is proposed by the XMP Specification Working Group consists of members from academia, research laboratories, and industries. It allows users to develop parallel applications and to tune performance with minimal and simple notation. A part of the design is based on experiences of High Performance Fortran (HPF) [7, 8] and Coarray Fortran (CAF) [9].

The features of XMP are as follows:

- XMP supports typical parallelization under "global-view model" programming and enables parallelizing the original sequential code using minimal modification with simple directives.
- XMP also includes a CAF-like PGAS feature as "local-view model" programming.
- The important design principle of XMP is "performance awareness". All actions of communication and synchronization are taken by directives or coarray syntax, different from HPF.
- XMP is defined as an extension for familiar languages, such as C and Fortran, to reduce code-rewriting and educational costs.

We have been developing an Omni XMP compiler as a prototype compiler. The Omni XMP compiler can compile an XMP C source

code and an XMP Fortran source code. Each source code and language are called XMP/C and XMP/Fortran in this paper. The Omni XMP compiler is available at the official website [10].

3. Implementation and Performance of benchmarks

3.1 Overview

This section provides a brief overview of XMP implementations and performances of the HPCC Benchmarks and the Himeno Benchmark. While the [6] also describes XMP implementations of the HPCC Benchmarks except for STREAM, an implementation and performance of FFT in this paper is improved compared to the [6].

All benchmarks were compiled using the Omni XMP compiler 0.7.0-alpha which will be available at SC13. In order to evaluate the performances of these benchmarks, we used 16,384 compute nodes at a maximum on the K computer which consists of 82,944 compute nodes. The specification of the K computer is shown in Table 2. All SLOCs of the implementations are excluded comments and blank lines, but included a validation operation and a printing performance result.

3.2 HPL

3.2.1 Differences from the implementation of last year

Firstly, we changed Column-major order into Row-major order for a coefficient matrix because of improving swap operation. Secondly, we gave an option "-mca coll_tuned_bcast_same_count 1" when executing an XMP HPL ($ mpiexec -mca coll_tuned_bcast_same_count 1 ./xmp_hpl). This option improves performance of MPI_Bcast() which is used in XMP library when size of sending data is the same on each process.

3.2.2 Implementation

We implemented the XMP HPL written in XMP/C. The SLOC of the XMP HPL is **306**.

The points of the implementation are as follows:

- Block-cyclic distribution

  Each node distributes a coefficient matrix *A[][]* in a block-cyclic manner, as is the case with hpcc-1.4 HPL. In the below 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 declares a global array *A[][]* and aligns *A[][]* with the template *t*. **Fig. 1**. shows the block-cyclic distribution in the below code.

```
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)
```

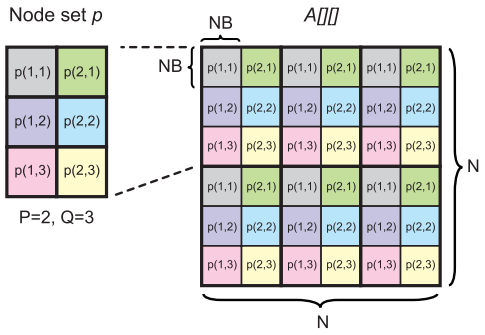

Fig. 1 Block cyclic distribution

- Panel broadcast by using **gmove** directive

  The below code and **Fig. 2** indicate a panel broadcast operation using the **gmove** directive and array section notation. The array *L[][]* is also distributed in the block-cyclic manner, but only the first dimension of the array *L[][]* is distributed. Thus, target elements of the array *A[j+NB:N-j-NB][j:NB]* (stripe block in Fig. 2) are broadcast to the array *L[j+NB:N-j-NB][0:NB]* that exists on each node.

3.2.3 Performance

We used the BLAS library which is parallelized with *pthread* automatically. We performed the XMP HPL with eight threads per process on one node. The size of the coefficient matrix *A[][]* is about 70% of the system memory. **Table 4** shows the performance and the theoretical peak performance of the system. For comparison, Table 4 also shows the performance of the XMP HPL of last year. The best performance is **542.94** TFlops in **65,536** CPU cores. This performance is approximately **53%** of the theoretical peak. The performances of this year are better than those of last year.

3.2.4 For more performance

We have implemented a new XMP HPL which uses coarray instead of **gmove** directive in the panel broadcast. The below code shows

```
1  double A[N][NB];
2  #pragma xmp align L[i][∗] with t(∗,i)
3    :
4  #pragma xmp gmove
5    L[j+NB:N−j−NB][0:NB] = A[j+NB:N−j−NB][j:NB];
```

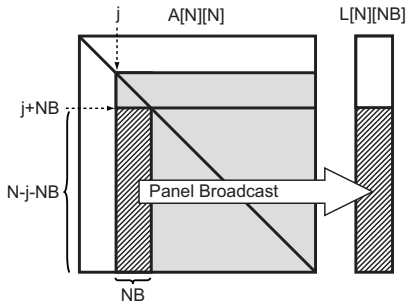

Fig. 2 Panel broadcast

Table 4 The performances of HPL

#Nodes	#Cores	Performance (TFlops)		of peak	
		XMP of this year	XMP of last year	**XMP of this year**	XMP of last year
1	8	**0.10**	0.08	**77.86%**	61.8%
4	32	**0.38**	0.30	**75.32%**	58.6%
16	128	**1.46**	1.15	**73.13%**	55.9%
64	512	**5.70**	4.33	**71.26%**	52.8%
256	2,048	**21.95**	15.74	**68.58%**	48.0%
1,024	8,192	**81.37**	52.93	**63.57%**	40.3%
4,096	32,768	**286.20**	156.50	**55.90%**	29.8%
8,192	65,536	**542.94**	(No Data)	**53.02%**	(No Data)

part of a code which performs “modified Increasing-ring” broadcast [13]. We think that this panel broadcast operation will improve the performance. Now we have been evaluating this code on the K computer. We hope to have a chance to present this result on BoF at SC13.

```
1  void row_bcast(int n, double c[n], int start, int len, int root){ // Q is a division number of row and P is a division number of column
2  #pragma xmp coarray c:[∗][Q]
3    int col_key = (xmp_node_num()−1)%P;
4    int row_key = (xmp_node_num()−1)/P;
5    int send = (P > col_key+1 ? col_key+1 : col_key+1−P);
6    int send2 = (root+2 < P ? root+2 : root+2−P);
7    int recv = (col_key−1 >= 0 ? col_key−1 : col_key−1+P);
8    int recv2 = (col_key−2 >= 0 ? col_key−2 : col_key−2+P);
9    if(recv2 == root) recv = recv2;
10
11   if(col_key == root){
12     c[start:len]:[send][row_key] = c[start:len];
13     if(send2 != col_key)
14       c[start:len]:[send2][row_key] = c[start:len];
15 #pragma xmp sync_memory
16 #pragma xmp post(p(send,row_key))
17 #pragma xmp post(p(send2,row_key))
18   } else{
19 #pragma xmp sync_memory
20 #pragma xmp wait(p(recv,row_key))
21     if(send != root && send2 != send){
22       c[start:len]:[send][row_key] = c[start:len];
23 #pragma xmp sync_memory
24 #pragma xmp post(p(send,row_key))
25     }
26   }
27 }
```

3.3 RandomAccess

3.3.1 Differences from the XMP implementation of last year

An XMP RandomAccess algorithm of this year is the same that of last year. However, in order to evaluate performance, we used 16,384 compute nodes at a maximum this year. By contrast, we used 8,192 compute nodes last year.

3.3.2 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 [11].

We implemented an algorithm with a set of remote writes to a coarray in local-view programming using XMP/C. 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's **post** and **wait** directives in order to realize asynchronous behavior of the algorithm.

The below code shows part of the XMP RandomAccess code. Line 1 declares arrays *recv[][]* and *send[][]* as coarrays. In line 18, 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 19, XMP/C extends the syntax of the array reference of the C language so that the "array section notation" can be specified instead of an index. The number before the colon in square brackets (*0*) indicates the start index of the section to be accessed, and the number after the colon (*nsend+1*) indicates its length. The number in square brackets after an array and the colon (*ipartner*) indicates the node number. Thus, line 19 means that elements from *send[isend][0]* to *send[isend][nsend]* are put to those from *recv[j][0]* to *recv[j][nsend]* in the *ipartner* node. In line 20, the **sync memory** directive is used to ensure the remote definition of a coarray is complete. In line 21 and 27, the **post** and **wait** directives are used for point-to-point synchronization. The **post** directive sends a signal to the node *ipartner* 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*.

The SLOC of the XMP RandomAccess is **250**.

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

The latest Omni XMP 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.

3.3.3 Performance

We performed the XMP RandomAccess, referred to as flat-MPI, on each CPU core. The table size is equal to 1/4 of the system memory. **Table 5** shows the performance results. 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 **163.63 GUP/s** (Giga UPdates per Second) for 131,072 CPU cores. Table 5 shows that the XMP RandomAccess and modified hpcc-1.4 have almost the same performances.

Table 5 The performances of RandomAccess

#Nodes	#Cores	Performance (GUP/s)	
		XMP	modified hpcc-1.4
1	8	**0.08**	0.08
8	64	**0.70**	0.62
64	512	**2.08**	2.41
512	4,096	**11.41**	13.55
4,096	32,768	**61.43**	75.08
8,192	65,536	**104.31**	120.24
16,384	131,072	**162.63**	(NO DATA)

3.4 FFT

3.4.1 Differences from the XMP implementation of last year

Firstly, we have developed and used an intrinsic transformational subroutine "XMP_TRANSPOSE()" to perform high-speed matrix transposition. Secondly, in order get better performance out of multi-core CPU, we have also used OpenMP pragma. Finally, we tune parameters, number of processes and length of vector, to reduce cache thrashing.

3.4.2 Implementation

We parallelized a subroutine "PZFFT1D0", which is the main kernel of the FFT. This subroutine is included in "pzfft1d.f" of the FFTE [2] optimized for the K computer. We have developed "xmp-pzfft1d.f90" based on "pzfft1d.f".

The below code shows part of an XMP FFT code in "xmp-pzfft1d.f90". This code is written in XMP/Fortran and OpenMP pragma. In line 1 to 8, the **node**, **template**, **distribute**, and **align** directives are describing the distribution of arrays in a block manner. In a six-step FFT, a matrix transposition must be performed before one-dimensional FFT. In the XMP FFT, the matrix transposition is implemented by the intrinsic transformational subroutine "XMP_TRANSPOSE()" in lines 10. The "XMP_TRANSPOSE()" performs all-to-all communication and a transposition of a local matrix in each node. When performing the communication, data pack/unpack operations are occurred internally. To improve performance, the data pack/unpack operations perform with thread-parallelization and cache-block tuning. In line 12 to 18, both XMP **loop** directive and OpenMP **parallel do** directive are used. Firstly, XMP **loop** directive parallelizes "do loop statements" in each process. Secondly, OpenMP **parallel do** directive parallelizes "do loop statements parallelized by XMP **loop** directive" in each thread.

The SLOC of the "xmp-pzfft1d.f90" is **70**. The SLOC of the "pzfft1d.f" in the FFTE is 101. Note that, we have also modified some files of the hpcc-1.4 FFT optimized for the K computer [2]. The total SLOC of the modified files is **70 + 169**. Moreover, the XMP FFT directly uses some files of the optimized hpcc-1.4 FFT. These files do not change. The total SLOC of the files of the hpcc-1.4 FFT is **283**. In addition, the XMP FFT uses other FFTE kernels and its C Interface. The total SLOC of the files of FFTE kernels and its C Interface is **1,892**. Hence, total SLOC is almost the same as the optimized hpcc-1.4 FFT, but the main kernel in "xmp-pzfft1d.f90" becomes simple.

```
1  !$XMP nodes p(*)
2  !$XMP template tx(NX)
3  !$XMP template ty(NY)
4  !$XMP distribute tx(block) onto p
5  !$XMP distribute ty(block) onto p
6  !$XMP align A(*,i) with ty(i)
7  !$XMP align B(*,i) with tx(i)
8  !$XMP align W(*,i) with tx(i)
9  ...
10 CALL XMP_TRANSPOSE(B,A,1)
11 ...
12 !$XMP loop on tx(I)
13 !$OMP parallel do
14 DO I=1,NX
15       DO J=1,NY
16             B(J,I)=B(J,I)*W(J,I)
17       END DO
18 END DO
19 ...
```

3.4.3 Performance

For using effectively cache on the K computer, we set parameters, the number of nodes and length of vector, which are not power of two. **Table 6** shows these parameters.

We performed the XMP FFT with eight threads per process on one node. For comparison, we also evaluated the modified hpcc-1.4 FFT which is optimized for the K computer. **Fig. 7** shows the performances. The best performance of the XMP FFT is **25,262.23 GFlops** (**24.67 TFlops**) for 73,728 CPU cores. The performance of the XMP FFT is almost the same as that of the hpcc-1.4 FFT. Moreover, for comparison, **Table 8** shows the performance of the XMP FFT of last year. Table 7 and Table 8 show that the performances of the XMP FFT of this year is better than that of last year.

Table 6 The parameters of FFT

#Nodes	Length of vector	Data size / System memory
36	6,635,520,000	34.3%
144	24,186,470,400	31.3%
576	99,532,800,000	32.2%
2,304	398,131,200,000	32.2%
9,216	1,528,823,808,000	30.1%

Table 7 The performances of FFT of this year

#Nodes	#Cores	Performance (GFlops)		of peak	
		XMP	modified hpcc-1.4	**XMP**	modified hpcc-1.4
36	288	**128.75**	134.50	**2.79%**	2.91%
144	1,152	**639.86**	677.34	**3.47%**	3.67%
576	4,608	**2,136.78**	2,211.21	**2.90%**	3.00%
2,304	18,432	**8,156.18**	8,426.56	**2.77%**	2.86%
9,216	73,728	**25,262.23**	25,546.65	**2.14%**	2.17%

Table 8 The performances of FFT of last year

#Nodes	#Cores	Performance (GFlops)		of peak	
		XMP	modified hpcc-1.4	**XMP**	modified hpcc-1.4
1	8	0.91	1.58	0.71%	1.23%
4	32	3.47	6.08	0.68%	1.19%
16	128	13.84	27.11	0.67%	1.32%
64	512	45.76	112.68	0.56%	1.37%
256	2,048	191.26	334.86	0.58%	1.02%
1,024	8,192	986.81	1,312.17	0.75%	1.00%

3.5 STREAM

3.5.1 Differences from the XMP implementation of last year

We did not implement STREAM last year.

3.5.2 Implementation

The below code shows part of an XMP STREAM code. The program is quite straightforward. Basically, a programmer only adds XMP directives into a sequential version STREAM. Line 8 defines XMP node set *p* to parallelize this program. Line 14 to 15 is a main kernel of STREAM which are the same as the original one. Line 42 gathers the performances of each process.

The SLOC of the XMP STREAM is 66.

```
1  #include <stdio.h>
2  #include <float.h>
3  #include <math.h>
4  #include <stdlib.h>
5  #include "xmp.h"
6  #define Mmin(a, b) (((a) < (b)) ? (a) : (b))
7  #define NTIMES 10
8  #pragma xmp nodes p(*)
9  ...
10 void HPCC_Stream(double *triadGBs, double *a, double *b, double *c, int size){
11   ...
12   for(k=0; k<NTIMES; k++) {
13     times[k] = -xmp_wtime();
14     for (j=0; j<size; j++)
15       a[j] = b[j] + scalar*c[j];
16
17     times[k] += xmp_wtime();
18   }
19
20   for (k=1; k<NTIMES; k++)
21     mintime = Mmin(mintime, times[k]);
22
23   curGBs = (mintime > 0.0 ? 1.0 / mintime : -1.0);
24   curGBs *= 1e-9 * 3 * sizeof(double) * size;
25   *triadGBs = curGBs;
26   ...
27 }
28
29 int main(int argc, char **argv){
30   double triadGBs, *a, *b, *c;
31
32   if(argc != 2){
33     if(xmp_node_num() == 1) fprintf(stderr, "./STREAM (number of vector)\n");
34     exit(1);
35   }
36   int size = atoi(argv[1]);
37   a = malloc(sizeof(double)*size);
38   b = malloc(sizeof(double)*size);
39   c = malloc(sizeof(double)*size);
40
41   HPCC_Stream(&triadGBs, a, b, c, size);
42 #pragma xmp reduction(+:triadGBs)
43
44   if(xmp_node_num() == 1)
45     printf("[Vector size is %d] Total Triad %.2f GB/s on %d process\n", size, triadGBs, xmp_num_nodes());
46
47   return 0;
48 }
```

3.5.3 Performance

We performed the XMP STREAM, referred to as flat-MPI, on each CPU core. The vector lengths of the arrays *a[]*, *b[]*, *c[]* is 67,108,864 which occupies 75 % of system memory on each compute node. For comparison, we also evaluated the hpcc-1.4 STREAM.

Table 9 shows the performances. The best performance of the XMP STREAM is **339,506.33 GB/s** (**331.55 TB/s**) for 65,536 CPU cores. Table 9 shows that the performance of the XMP STREAM is a little better that that of hpcc-1.4.

Table 9 The performances of STREAM

#Nodes	#Cores	Performance (GB/s)		of peak	
		XMP	hpcc-1.4	**XMP**	hpcc-1.4
1	8	**32.25**	34.97	**50.39%**	54.64%
4	32	**159.66**	146.00	**62.37%**	57.03%
16	128	**638.41**	566.46	**62.34%**	55.31%
64	512	**2,698.31**	2,243.22	**65.88%**	54.77%
256	2,048	**10,088.73**	9,276.53	**61.58%**	56.62%
1,024	8,192	**44,251.13**	37,355.21	**67.52%**	57.00%
4,096	32,768	**176,763.33**	149,837.21	**67.43%**	57.16%
8,192	65,536	**339,506.33**	309,158.42	**64.76%**	58.97%

3.6 Himeno Benchmark

3.6.1 Differences from the XMP implementation of last year

The algorithm of the XMP Himeno Benchmark of this year is the same as that of last year. However performance of **reflect** directive of the Omni XMP compiler is improved to synchronize data of overlapped region [3]. Moreover, we tune a parameter "width of overlapped region" to use cache effectively.

3.6.2 Implementation

The Himeno Benchmark evaluates performance of incompressible fluid analysis code using the Jacobi iteration method. This benchmark measures performance to proceed major loops in solving the Poisson's equation solution in Flops. To verify the result of this benchmark, it calculates the residual of the Jacobi iteration method. The reason of selecting this benchmark is a good example of a stencil application benchmark and to demonstrate parallelization by XMP **shadow** and **reflect** directives which communicate and synchronize the overlapped regions.

The below code shows part of the XMP Himeno Benchmark code written in XMP/Fortran. In line 5, a **shadow** directive declares an overlapped region of the distributed array *p*. The **shadow** directive specifies the width of the overlapped region. In line 7 and 22, the **reflect** directive synchronizes data of the overlapped region onto the neighboring process before referring array *p* by loop iteration. This parallelization is very simple and straightforward. Basically, a programmer only add XMP directives into a sequential version Himeno Benchmark.

The SLOC of XMP HIMENO Benchmark is **137** where nineteen XMP directives are used. Hence the SLOC of the sequential version Himeno Benchmark is **118**. Besides, the SLOC of the original Himeno Benchmark is **380** which is written in MPI Fortran.

```
1  !$xmp nodes n(2,2)
2  !$xmp template t(mimax,mjmax,mkmax)
3  !$xmp distribute t(*,block,block) onto n
4  !$xmp align (*,j,k) with t(*,j,k) :: p, bnd, wrk1, wrk2
5  !$xmp shadow p(0,2:1,2:1)
6  ...
7  !$xmp reflect (p)
8  DO loop = 1, nn
9       GOSA = 0.0
10 !$xmp loop (J,K) on t(*,J,K)
11      DO K = 2, kmax-1
12           DO J = 2, jmax-1
13                DO I = 2, imax-1
14                     S0 = a(I,J,K,1)*p(I+1,J,K) + ...
15                     SS = (S0*a(I,J,K,4)-p(I,J,K))*bnd(I,J,K)
16                     GOSA = GOSA + SS * SS
17                       ...
18                enddo
19           enddo
20      enddo
21      ...
22 !$xmp reflect (p)
23 !$xmp reduction (+:GOSA)
24 enddo
```

The synchronization of the overlapped region in **reflect** directive internally needs data pack/unpack operations. In order to obtain better performance, the Omni XMP compiler performs these operations with thread-parallelization [3].

3.6.3 Performance

For using cache on the K computer efficiently, we set **p(0,2:1,2:1)** in Line 5 as widths of the overlapped region. This line means that the overlapped region is added to left and right sides of array *p()*. The width of the left side is 2, and that of the right side is 1 in second and third dimensions of array *p()*. In fact, the Himeno Benchmark needs only 1 element in both sides of array *p()*. However, it is possible to reduce cache thrashing by setting odd widths in one side.

We set the number of elements of array $p(128, 512 \times \sqrt{n}, 512 \times \sqrt{n})$, n is a number of processes for weak scaling. For thread-parallelization of the XMP Himeno Benchmark, we utilized the automatic thread-parallelization feature provided by the Fujitsu Fortran Compiler. We performed the XMP Himeno Benchmark with eight threads per process on one node. For comparison, we also evaluated the original Himeno Benchmark [1]. **Table 10** shows the performances. Moreover, for comparison, Table 10 shows the performance of the XMP Himeno Benchmark of last year.

Table 10 The performances of Himeno Benchmark

#Nodes	#Cores	Performance (GFlops)			of peak		
		XMP of this year	Original	XMP of last year	**XMP of this year**	Original	XMP of last year
1	8	**19.00**	19.51	12.43	**14.84%**	15.24%	9.71%
4	32	**67.33**	61.72	39.78	**13.15%**	12.05%	7.77%
16	128	**300.14**	248.59	127.81	**14.66%**	12.14%	6.24%
64	512	**1,200.83**	1,063.19	342.82	**14.66%**	12.98%	4.18%
256	2,048	**4,799.32**	3,962.69	1,495.55	**14.65%**	12.09%	4.56%
1,024	8,192	**19,175.58**	15,855.84	5,773.18	**14.63%**	12.10%	4.40%
4,096	32,768	**76,631.04**	63,432.90	(NO DATA)	**14.62%**	12.10%	(NO DATA)
16,384	131,072	**306274.34**	253730.23	(NO DATA)	**14.60%**	12.10%	(NO DATA)

The best performance of the XMP Himeno Benchmark is **306274.34 GFlops** (**299.10 TFlops**) for 131,072 CPU cores. The result in Table 10 indicates that the performance of XMP is better than those of the original and last year. The reason is that XMP Himeno Benchmark performs data pack/unpack operations for synchronization of the overlap region with thread-parallelization.

4. Conclusion

This report has investigated the productivity and the performance of the XMP PGAS language through the HPCC and the Himeno Benchmarks. XMP has a rich set of features based on global-view and local-view model that allows users to develop applications with a little cost.

Acknowledgment

The modification of the two functions for sorting and updating local table in RandomAccess were done in cooperation with Fujitsu Limited. Moreover, we gratefully thank Ikuo Miyoshi who belongs to Fujitsu Limited for his valuable lecture of HPL. The study was supported by the “Feasibility Study on Future HPC Infrastructure” project funded by the Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] The Riken Himeno CFD Benchmark. http://accc.riken.jp/2444.htm
[2] Daisuke Takahashi, Atsuya Uno, and Mitsuo Yokokawa. “An Implementation of Parallel 1-D FFT on the K computer”, IEEE 14th International Conference on High Performance Computing and Communications, 2012
[3] 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.
[4] 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.
[5] Jinpil Lee. “A Study on Productive and Reliable Programming Environment for Distributed Memory System”, March, 2012.
[6] 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.
[7] C.H. Koelbel, D.B. Loverman, R. Shreiber, GL. Steele Jr., and M.E. Zosel. “The High Performance Fortran Handbook”, MIT Press, 1994.
[8] 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
[9] R. Numwich and J. Reid. “Co-Array Fortran for parallel programming”. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.
[10] Omni XcalableMP Compiler. http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/xcalablemp/
[11] R. Ponnusamy, A. Choudhary and G. Fox. “Communication Overhead on CM5: An Experimental Performance Evaluation”, Proc. Frontiers ’92, pp.108–115, 1992.
[12] HPC Challenge Website. http://icl.cs.utk.edu/hpcc/software/index.html
[13] HPL Algorithm. http://www.netlib.org/benchmark/hpl/algorithm.html