# Productivity and Performance of the HPC Challenge Benchmarks with the XcalableMP PGAS Language

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

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

**Abstract**

The present paper introduces designs of the XcalableMP PGAS language for improved productivity and performance of a high performance computing (HPC) system. The design of a unique XcalableMP programming model is based on both local-view and global-view models. This design allows programmers to easily develop HPC applications. Moreover, in order to tune HPC applications, XcalableMP provides inquiry functions for programmers to obtain local memory information of a global array. In the present paper, we evaluate the productivity and the performance of XcalableMP through implementations of the High-Performance Computing Challenge (HPCC) Benchmarks. We describe the implementations of three HPCC benchmarks, including RandomAccess, High-performance Linpack (HPL), and Fast Fourier Transform (FFT). In order to evaluate the performance of XcalableMP, we used the K computer, which is a leadership-class HPC system. As a result, we achieved 163 GUPS with RandomAccess (using 131,072 CPU cores), 543 TFlops with HPL (using 65,536 CPU cores), and 24 TFlops with FFT (using 262,144 CPU cores). These results reveal that XcalableMP has good performance on the K computer.

## 1 Introduction

At present, leadership-class high-performance computing (HPC) systems consist of hundreds of thousands of compute nodes, and the number of the compute nodes will continue to increase. Applications run on the system are used so many times that programmers require a high-performance programming language to reduce the execution time. Moreover, in order to reduce the programming and maintenance costs, programmers also require the programming language to be productive. Partitioned Global Address Space (PGAS) model languages, such as SHMEM[12], Global Arrays[23], Coarray Fortran[24], Titanium[29], Unified Parallel C[13], Chapel[11], and X10[25], are emerging as alternatives to Message Passing Interface (MPI)[26]. In particular, the greater part of Coarray Fortran is incorporated into the Fortran 2008 standard. In addition to these languages, we have been designing and developing a new PGAS model language called XcalableMP (XMP) [19, 21, 22, 28], which is a directive-based language extension of C and Fortran.

Since most existing HPC applications are written in MPI, PGAS languages are not widely used. One reason for this is that the productivity and performance of PGAS languages on an HPC system remain unclear. Therefore, in the present study, we reveal the productivity and performance of the XMP PGAS language using a maximum of 32,768 compute nodes (262,144 CPU cores) of the K computer. The K computer was ranked 4th in the Top500[9] on June, 2013. In order to measure the productivity and performance, we have used the High-Performance Computing Challenge (HPCC) benchmarks[5]. The HPCC benchmarks are a set

of benchmarks to evaluate multiple attributes, such as the random access speed of memory, system interconnect, and the processor on an HPC system. The HPCC benchmarks are also used at the HPCC Award Competition, which has been held at the supercomputer conference since 2005. The HPCC Award Competition consists of two classes. The class 1 competition uses four of the HPCC benchmarks, namely, RandomAccess, High-performance Linpack (HPL), Fast Fourier Transform (FFT), and STREAM, to evaluate the overall performance of an HPC system. These benchmarks are frequently used in scientific fields. The class 2 competition uses three or four of the HPCC benchmarks to evaluate the productivity and performance of a programming language. In the present paper, we have implemented RandomAccess, HPL, and FFT benchmarks because parallelizing the STREAM benchmark is easy. Through these implementations and evaluations, we investigate potential improvements in HPC systems that may be achieved through proper XMP design.

To this end, the present paper makes the following specific contributions: (1) Designs of the XMP programming model for an HPC system are proposed and their effectiveness is validated; and (2) The implementation of the HPCC benchmarks using XMP and the tuning of these benchmarks are presented. These contributions are also useful for other PGAS languages.

The remainder of the present paper is structured as follows. Section 2 presents the requirements of a promising programming model on an HPC system. Section 3 introduces an overview of XMP. Section 4 describes implementations and evaluations of the HPCC benchmarks. Section 5 discusses the productivity and performance of the HPCC benchmarks in XMP. Section 6 summarizes the present paper and describes areas for future research.

# 2 Requirements of a Programming Model on High-performance Computing System

A promising programming model for HPC systems must have both high productivity and high performance. In this section, we consider the features necessary for a promising programming model.

## 2.1 High Productivity

### 2.1.1 Components of Productivity

The productivity of HPC applications is based on not only the programming cost but also the porting, tuning, maintenance, and educational costs. In addition, the productivity is related to the ability to easily reuse and extend a part of the application. Therefore, the productivity should not be evaluated based solely on the Source Lines Of Code (SLOC), which should be one of indicators used in the evaluation of the productivity.

### 2.1.2 Performance Portability

In general, the lifetime of an HPC application is longer than the lifetime of the hardware. Once the HPC application has been created, the application is used for several decades while maintaining, expanding, and changing its data structure and/or algorithm. Therefore, the HPC application should enable high performance on different machines with little effort. In order to meet this requirement, the source code must clearly indicate the behavior of the computer, for example, access to local memory and data communication.

#### 2.1.3 Usage of Numerical Libraries

In order to reduce the porting and programming costs and allow high performance, HPC applications often use numerical libraries such as BLAS[1] and Scalapack[2]. Therefore, a promising language must be able to use the numerical libraries directly or to use a language-specific library such as UPCBLAS[16]. In order to use numerical libraries directly from the PGAS model language, the language should expose local memory information of its global address space.

#### 2.1.4 Coexistence with MPI

At present, most HPC applications are written in MPI. Since the performances of existing MPI applications are extremely good, it is not worth rewriting all of the existing MPI applications in terms of a promising language. Therefore, the promising language and MPI should coexist. In particular, the promising language must be able to call an MPI library and an MPI application. Moreover, a library and an application written in the promising language must be able to be called from an MPI application. In order to meet these requirements, the promising language must deal with MPI objects such as an MPI communicator.

### 2.2 High Performance

#### 2.2.1 Ease of Tuning

In order to tune high-performance applications, programmers must perform hardware-aware programming. In other words, each line of the promising language allows programmers to clearly understand what happens on the hardware.

#### 2.2.2 Support of Any Parallelizations

In order to increase the performance/cost ratio, the commonly used HPC system is a multicore cluster and may also have accelerators such as a Graphics Processing Unit (GPU) or a Many Integrated Core (MIC). Therefore, the promising language should control hybrid-parallelization, which combines process-parallelization with thread-parallelization, and parallelization on accelerators. At present, MPI and OpenMP are frequently used for hybrid parallelization, and CUDA and OpenACC[7] are frequently used for parallelization on accelerators. Note that the promising language does not need to support all parallelizations as its function. For example, if the promising language can use OpenMP directives, the promising language does not have to support thread-parallelization as its function.

## 3 Overview of XcalableMP

In this section, we describe some of the features of the design of XMP related to performance and productivity, as described in Section 2. In addition, we introduce the implementation of an XMP compiler.

### 3.1 Design

The XMP specification[28] has been designed by the XMP Specification Working Group, which consists of members from academia, research laboratories, and industries. This Working Group is supported by the PC Cluster Consortium in Japan[8].

**XMP/Fortran**

| | |
|---|---|
| integer function | xmp_array_gtol(*d*, *g_idx*, *l_idx*) |
| type(xmp_desc) | *d* |
| integer | *g_idx(NDIMS)* |
| integer | *l_idx(NDIMS)* |

**XMP/C**

| | |
|---|---|
| void | xmp_array_gtol(xmp_desc_t *d*, int *g_idx[]*, int *l_idx[]*) |

Figure 1: Example of XMP Inquiry Functions

```
 1  #include "mpi.h"
 2  #include "xmp.h"
 3  #pragma xmp nodes p(4)
 4
 5  void main(int argc, char *argv[]) {
 6    xmp_init_mpi(argc, argv);
 7    MPI_Comm comm = xmp_get_mpi_comm();
 8
 9    user_mpi_func(comm);
10
11    xmp_finalize_mpi();
12  }
```

Figure 2: Example of the Use of the MPI Programming Interface in XMP/C

### 3.1.1 Familiar HPC Language (related to Section 2.1.1)

In order to reduce programming and educational costs, XMP is extended to familiar HPC languages, such as Fortran and C, and these extensions are referred to herein as XMP/Fortran and XMP/C, respectively.

### 3.1.2 Support of Global-view and Local-view (related to Section 2.1.1)

In order to develop various applications, the XMP programming model is an SPMD under the global-view and local-view models. In the global-view model, XMP enables parallelization of an original sequential code using minimal modification with simple directives such as OpenMP. In the local-view model, the programmer can use coarray syntax in both XMP/Fortran and XMP/C. In particular, XMP/Fortran is designed to be compatible with Coarray Fortran.

### 3.1.3 XMP Inquiry Functions (related to Sections 2.1.2, 2.1.3, and 2.2.1)

In order to use numerical libraries, XMP inquiry functions provide local memory information of a global array defined in the XMP global-view model. Figure 1 shows one of the XMP inquiry functions `xmp_array_gtol()`. This function translates an index (specified by *g_idx*) of a global array (specified by descriptor *d*) into the corresponding index of its local section and sets to an array (specified by *l_idx*). In addition to this function, the XMP specification defines inquiry functions that enable the collection of other local memory information, such as the pointer, size, and leading dimension of a global array. The XMP inquiry functions enable the use of numerical libraries and tune the application using the local memory information.

### 3.1.4 MPI Programming Interface (related to Section 2.1.4)

In order to call an MPI program from an XMP program, the MPI programming interface, a function `xmp_get_mpi_comm()` in XMP/C, is provided. This function returns a handle of an MPI communicator associated with the execution of processes. In addition to this function, the XMP specification defines two functions, `xmp_init_mpi()` and `xmp_finalize_mpi()`, to initialize and finalize the MPI execution environment in XMP/C. Figure 2 shows an example of using these functions. Line 2 includes xmp.h, in which the functions are defined. Line 3 defines the XMP node set, which is a process unit and typically corresponds to an MPI process. Thus, this program is executed on four nodes. Lines 6 and 11 perform initialization and finalization of the MPI execution environment. Line 7 obtains an MPI communicator, and Line 9 sends the MPI

communicator to the user-defined MPI function. Similarly, XMP/Fortran also provides these functions.

The XMP specification has not yet provided a function of how to call an XMP program from an MPI program. However, in the implementation of FFT using XMP (described in Section 4.3.1), an FFT main kernel is called by an MPI program (Figure 13) because an Omni XMP compiler informally supports the function. Now, we are planning to design a new XMP specification to support the function.

#### 3.1.5 Leverage HPF's Experience (related to Section 2.2.1)

Part of the design of XMP is also based on High Performance Fortran (HPF)[17, 18]. Thus, some concepts of HPF, such as the use of a template, which is a virtual global index, have been inherited. However, all communication and synchronization actions occurred only at points of the XMP directive or coarray syntax because of performance awareness. In other words, an XMP compiler does not automatically insert communication calls. This design is different from HPF. Since it is clear what each line does in XMP, this design enables programmers to easily tune an application.

#### 3.1.6 Use of Other Parallelizations (related to Section 2.2.2)

In XMP, OpenMP and OpenACC pragmas can also be used only for local calculations because programmers can obtain the local memory information (described in Section 3.1.3). We have been designing a new XMP specification for mixing communication with these pragmas.

On the other hand, XMP-dev[19, 20], which is an extended XMP, has been proposed. Since XMP-dev provides pragmas for accelerators, XMP-dev allows programmers to easily develop parallel applications on an accelerator cluster.

### 3.2 Implementation

We have been developing an Omni XMP compiler[6, 19] as a prototype compiler to compile XMP/C and XMP/Fortran codes. The Omni XMP compiler is a source-to-source compiler that translates an XMP/C or XMP/Fortran code into a parallel code using an XMP runtime library. The parallel code is compiled by the native compiler of the machine (e.g., Intel compiler).

The latest Omni XMP compiler (version 0.6.2-beta) has been optimized for the K computer. For example, 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 [15]. In Section 4, we use the Omni XMP Compiler to evaluate the HPCC benchmarks.

## 4 High-Performance Computing Challenge Benchmarks in XcalableMP

In this section, we introduce the proposed implementations of the HPCC benchmarks and describe the evaluations thereof. In order to evaluate the performance of these implementations, we used a maximum of 32,768 compute nodes (262,144 CPU cores) of the K computer (CPU: SPARC64 VIIIfx 2.0 GHz (eight CPU cores), Memory: DDR3 SDRAM 16 GB 64 GB/s, Network: Torus fusion six-dimensional mesh/torus network 5 GB/s $\times$ 2 (bi-directional) $\times$ 10 (links per node)). In addition, in Section 4.4, we evaluate the performance of these implementations on a general PC cluster.

An overview of each benchmark is presented below.

- The RandomAccess benchmark measures the performance of random integer updates of memory via interconnect.
- The HPL benchmark measures the floating point rate of execution to solve a dense system of linear equations using LU factorization.
- The FFT benchmark measures the floating point rate of execution for double-precision complex one-dimensional Discrete Fourier Transform.

## 4.1 RandomAccess

### 4.1.1 Implementation

The proposed algorithm is iterated over sets of CHUNK updates on each node. In each iteration, the proposed 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 [14].

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.

Figure 3 shows part of the proposed RandomAccess code. Line 1 declares arrays *recv[][]* and *send[][]* as coarrays. Since these arrays on both hand sides of a coarray operation should be coarrays due to a restriction of the interface of the K computer, the array *send[][]* on the right-hand side, which was originally a normal (non-coarray) data, is declared as a coarray. 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*.

### 4.1.2 Performance

We performed the proposed implementation, referred to as flat-MPI, of RandomAccess on each CPU core. The table size is equal to 1/4 of the system memory. Figure 4 shows the performance results. For comparison, we evaluated the modified hpcc-1.4.2[5] RandomAccess, for which the functions for sorting and updating the table are specifically optimized for the K computer. The best performance of the XMP implementation is 163 GUPS (Giga UPdates per Second)

```
 1 u64Int recv[MAXLOGPROCS][RCHUNK+1]:[*],
           send[2][CHUNKBIG+1]:[*];       // Declare Coarrays
 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, ...);
```

Figure 3: Source Code of RandomAccess

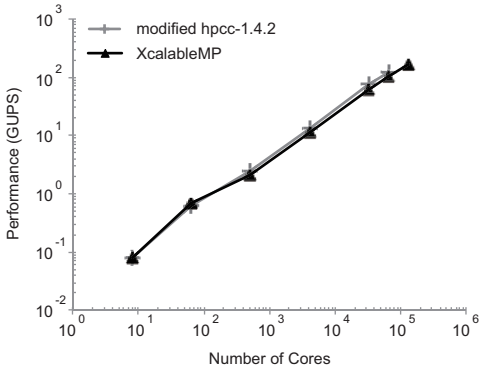



Figure 4: Performance of RandomAccess on the K Computer

```
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)
```

Figure 5: Define the Distribution Array for High-performance Linpack

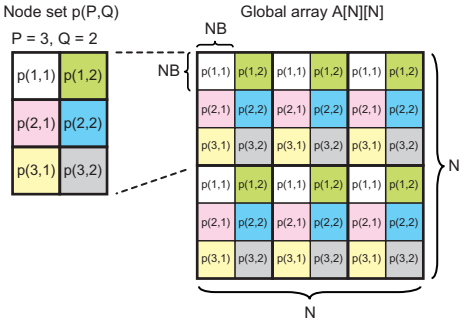



Figure 6: Block-cyclic Distribution in High-performance Linpack

in 131,072 CPU cores. Figure 4 shows that the XMP implementation and modified hpcc-1.4.2 have approximately the same performances.

## 4.2 High-performance Linpack

### 4.2.1 Implementation

Figure 5 shows part of the proposed HPL implementation in XMP/C, where each node distributes a global array *A[][]* in a block-cyclic manner, as is the case with hpcc-1.4.2 HPL. In this code, a **template** directive declares a two-dimensional template *t*, and a **node** directive

```
1 double L[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];
```

Figure 7: Source Code of a **gmove** Directive in High-performance Linpack

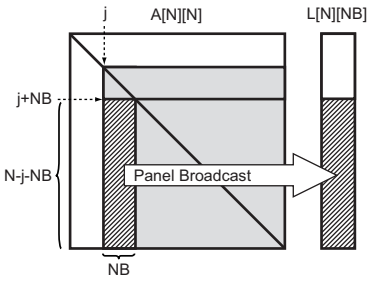



Figure 8: Panel Broadcast Using a **gmove** Directive in Figure 7

```
1  #include "xmp.h"
2  int g_idx[2], l_dx[2];
3  ...
4  // set g_idx
5  ...
6  xmp_desc_t A_desc = xmp_desc_of(A); // A is a global array
7  xmp_array_gtol(A_desc, g_idx, l_idx);
8  int local_y = l_idx[0];
9  int local_x = l_idx[1];
10 cblas_dgemm(..., N/Q-local_y, N/P-local_x, ...,
           &L[g_idx[0]][0], ..., &A[g_idx[0]][g_idx[1]], ...);
```

Figure 9: Example of a Calling BLAS Library

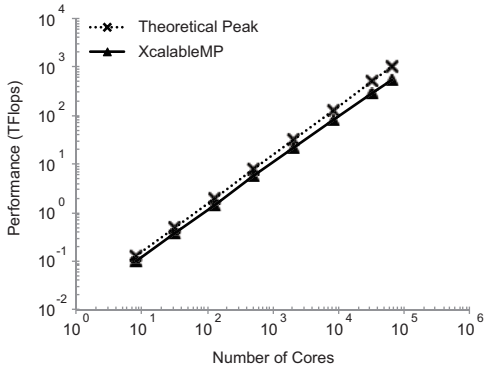



Figure 10: Performance of High-performance Linpack on the K Computer

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 $t$. Figure 6 shows the block-cyclic distribution in Figure 5.

Figures 7 and 8 indicate a panel broadcast operation in HPL 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 Figure 8) are broadcast to the array *L[j+NB:N-j-NB][0:NB]* that exists on each node. Note that since a non-blocking "gmove" directive has not yet been implemented in the Omni XMP compiler, this "gmove" directive is a blocking operation.

In HPL, the performance of matrix multiply is significant important. Thus, high-performance *DGEMM()* and *DTRSM()* functions in the BLAS library should be used. In order to use the BLAS library optimized for the K computer, we call *DGEMM()* and *DTRSM()* functions using XMP inquiry functions (refer to Section 3.1.3). Figure 9 shows part of a code for calling the BLAS library. Line 1 includes the header file "xmp.h" to use XMP inquiry functions. Line 6 gets the descriptor of the global array *A[][]*, and Line 7 sets the local indices from global indices. Line 10 calls the DGEMM function using local indices. The *N/Q-local_y* and *N/P-local_x* are the local widths of the column and row of the calculated matrixes. In

```
 1 COMPLEX*16 :: A(NX,NY), A_WORK(NX,NY), B(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 !$XMP align A_WORK(i,*) with tx(i)
 9 !$XMP align B(*,i) with tx(i)
10   ...
11 !$XMP gmove
12 A_WORK(1:NX,1:NY) = A(1:NX,1:NY)
13
14 !$XMP loop (i) on tx(i)
15 !$OMP parallel do
16 DO 70 I=1,NX
17    DO 60 J=1,NY
18       B(J,I)=A_WORK(I,J)
19    60 CONTINUE
20 70 CONTINUE
```

Figure 11: Source Code of the Fast Fourier Transform

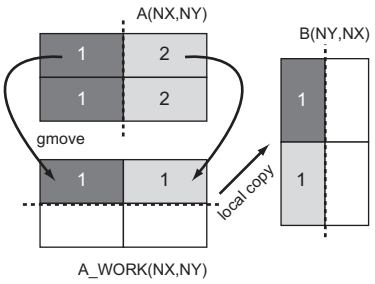



Figure 12: Matrix Transpose in the Fast Fourier Transform

addition, XMP has a rule that a pointer of a global array indicates a local pointer on the node to which it is distributed. Thus, *&L[g_idx[0]][0]* and *&A[g_idx[0]][g_idx[1]]* indicate proper local pointers. In contrast to the hpcc-1.4.2 HPL, the *DGEMM()* function is called only one time in each update operation, because a node has all of the data for the matrix multiply.

#### 4.2.2 Performance

We performed the proposed implementation with eight threads per process on one node. The size of the array *A[][]* is approximately 90% of the system memory. Figure 10 shows the performance and the theoretical peak performance of the system. The best performance is 543 TFlops in 65,536 CPU cores. This performance is approximately 53% of the theoretical peak. The performance of using only eight CPU cores (on one node) is 100 GFlops, which is approximately 78% of the theoretical peak. The parallelization efficiency appears not to be very good. The main reason is that an effective panel broadcast operation has not yet been implemented. We consider this issue in Section 5.1.2.

### 4.3 Fast Fourier Transform

#### 4.3.1 Implementation

We parallelized a subroutine, "PZFFT1D0", which is the main kernel of the FFT. This subroutine is included in ffte-5.0 [3]. Figure 11 shows part of the proposed FFT implementation in XMP/Fortran. In Lines 2 through 9, the **template**, **nodes**, **distribute**, and **align** directives describe the distribution of arrays *A()*, *A_WORK()*, and *B()* in a block manner.

In a six-step FFT, which is an algorithm used in ffte-5.0, a matrix transpose operation must be performed before one-dimensional FFT. In the sequential version FFT, the matrix transpose is implemented by local memory copy between *A()* and *B()*. In the XMP version, the matrix transpose operation is implemented by the **gmove** directive and local memory copy in Lines 11 through 20. Figure 12 shows how to transpose matrix *A()* to *B()*. In Figure 12, the number is the node number to which the block is allocated, and the dotted lines indicate how to distribute the matrices. Since the distribution manner of the arrays (*A()* and *B()*) is different, node 1

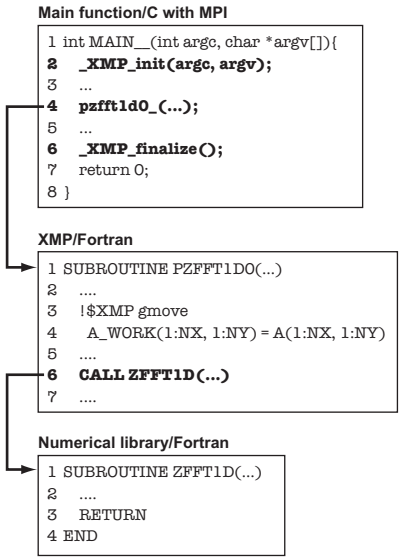



Figure 13: Example of Calling an XMP Program from an MPI Program

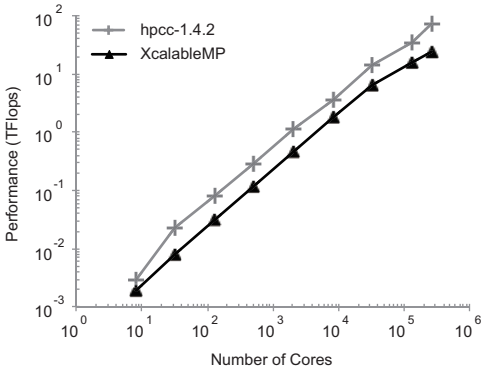


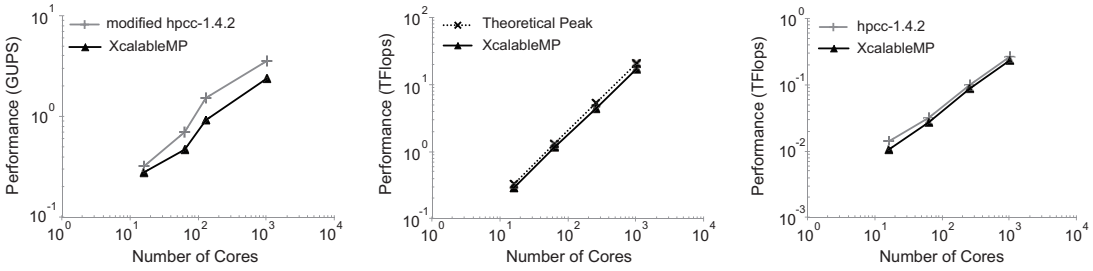

Figure 14: Performance of the Fast Fourier Transform on the K computer

does not have all of the elements of matrix *A()*, which are needed for the transpose. The **gmove** directive is used to collect these elements. Initially, a new array *A_WORK()* is declared to store the elements. *A_WORK()* is distributed by template *tx*, which is used to distribute *B()*. Consequently, the local block of *A_WORK()* and that of *B()* have the same shape. By the all-to-all communication of the **gmove** directive in Lines 11 and 12, all elements needed for transpose are stored in *A_WORK()*. Finally, Lines 14 through 20 copy the elements to *B()* using the **loop** directive and OpenMP thread-parallelization. The **loop** directive distributes iterations that are specified by the on clause and are executed in parallel.

This FFT implementation is a good demonstration of how to mix an XMP program with an MPI program. The subroutine "PZFFT1D0" written in XMP/Fortran is called by the main function written in the C language with MPI. In addition, the "PZFFT1D0" calls other subroutines in ffte-5.0. Figure 13 shows these procedures, which are the same as the hpcc-1.4.2 FFT. Lines 2 and 6 in "Main function/C" of Figure 13 call the `_XMP_init()` and `_XMP_finalize()` functions to initialize and finalize an XMP execution environment. These functions are used internally in the Omni XMP compiler and also set up an MPI execution environment. Since this method is informal, we intend to design official functions for how to call an XMP program from another language.

### 4.3.2 Performance

We performed the proposed implementation with eight threads per process on one node. For comparison, we evaluated the hpcc-1.4.2 FFT. The program size is approximately 60% of the system memory. Figure 14 shows the performance of the implementations. The best performance of the XMP implementation is 24 TFlops for 262,144 CPU cores. The performance of the XMP implementation is approximately half that of the hpcc-1.4.2 FFT. The primary reason for this difference is that an internal packing operation for arrays is slow when using the **gmove** operation. We consider this issue in Section 5.1.3.




(a) RandomAccess (b) High-performance Linpack (c) Fast Fourier Transform

Figure 15: Performance Results on HA-PACS

## 4.4 Performance on a General PC Cluster

In this section, we describe the performance of the proposed implementations on a general PC cluster. We used HA-PACS (CPU: Intel Xeon E5-2670 2.60 GHz [eight CPU cores x two sockets/node], memory: DDR3 SDRAM 128 GB 51.4 GB/s/socket, network: Fat-Tree Infiniband QDR 4 GB/s x 2 (bi-directional) x two rails)[4] located at the University of Tsukuba. We used a maximum of 64 compute nodes (1,024 CPU cores) in this evaluation.

### 4.4.1 RandomAccess

We performed the RandomAccess on each of the CPU cores (flat-MPI). Figure 15(a) shows that the performance of XMP RandomAccess is slightly worse than that of modified hpcc-1.4.2. The reason for the low performance is the use of GASNet[10] as a one-sided communication layer for a coarray on a general PC cluster. Thus, the difference in performance between GASNet and MPI is less than that between the extended RDMA and MPI on the K computer.

### 4.4.2 High-performance Linpack

We performed the HPL on each socket (eight threads/process, two process/node). Figure 15(b) shows the performances of HPL are approximately the same. Using 1,024 CPU cores (128 processes, 64 compute nodes), the performance of XMP implementation is 81% of the theoretical peak performance of the system. The reason for the good performance is that a compute node of HA-PACS has more memory than that of the K computer. Thus, the ratio of computation on HA-PACS is larger than that on the K computer, and the difference between the performances is slight.

### 4.4.3 Fast Fourier Transform

We performed FFT on each of the CPU cores. The reason for using flat-MPI was that the performance of flat-MPI was better than that of hybrid-parallelization. Figure 15(c) shows that both the performances of the FFT are almost the same. The reason for this is that thread-parallelization for the packing array in the hpcc-1.4.2 FFT was not performed in this evaluation (refer to Section 5.1.3).

# 5 Consideration

## 5.1 Productivity and Performance of XcalableMP

In this section, we discuss the productivity and performance of XMP for the implementations of the HPCC benchmarks in Section 4.

### 5.1.1 RandomAccess

In general, a coarray is a more convenient means for expressing communications in parallel programs than traditional MPI library routines. Therefore, it can be said that XMP provides good productivity in implementing the RandomAccess benchmark.

On the other hand, the XMP version with coarrays does not outperform the MPI, even though the extended RDMA interface used to implement coarrays in XMP could make the most use of the underlying network hardware and be faster than the vanilla MPI. There are two reasons for this. First, there is no way to complete memory operations from/to a certain image (or node in XMP) in the Fortran standard, and the **sync memory** statement being used, which completes all memory operations on the image, is too strong for such a purpose. Second, the current implementations of the **post** and **wait** directives for point-to-point synchronization are not optimal and could be implemented more efficiently with the extended RDMA interface.

### 5.1.2 High-performance Linpack

The **gmove** directive is useful for increasing the productivity. The panel broadcast operation is performed while maintaining the global image in Figures 7 and 8. In other words, the **gmove** directive automatically performs packing/unpacking data and communication for global array. However, it is difficult for programmers to understand in detail what operations and communications occur.

In order to increase the parallel efficiency of the HPL, it is known that scheduling of communication for panel broadcast is significant important. For example, a panel broadcast algorithm provided by the hpcc-1.4.2 HPL first sends the panel to a right-hand neighbor process, because the process can start a calculation using the received panel as soon as possible. If XMP uses this algorithm, XMP will use the function of the XMP coarray. In this case, the performance of the HPL will increase, but the complexity of the code will also increase.

### 5.1.3 Fast Fourier Transform

The SLOC of the subroutine "PZFFT1D0" written in XMP/Fortran is 65. On the other hand, the SLOC of an original "PZFFT1D0" written in Fortran with MPI is 101. Moreover, the code of the "PZFFT1D0" becomes simple because this code maintains a global image using the **gmove** and **loop** directives in Figure 11.

However, the main reason for the low performance is the **gmove** directive. Before the all-to-all communication, the transported data (two-dimensional array *A()* in Figure 12) must be packed in a one-dimensional array. This operation is performed internally in the **gmove** directive. In the hpcc-1.4.2 FFT, this operation is also performed in the same manner as the XMP version FFT. However, the hpcc-1.4.2 FFT performs this operation using thread-parallelization. The **gmove** directive has not yet supported internal thread-parallelization.

Table 1: Performance and SLOC of HPCC Awards Class 2 and XcalableMP in 2011 and 2012

| Language | Machine | RandomAccess (GUPS) [SLOC] | HPL (TFlops) [SLOC] | FFT (TFlops) [SLOC] |
|---|---|---|---|---|
| X10 | IBM Power 775 | 844 [143] | 589 [708] | 35 [236] |
| Chapel | Cray XE6 | 4 [112] | 8 [658] | (NO DATA) |
| Charm++ | BlueGene/P & Cray XT5 | 43 [138] | 55 [1,770] | 3 [185] |
| Coarray Fortran | Cray XT4 & XT5 | 2 [409] | 18 [786] | 0.3 [450] |
| XcalableMP | The K computer | 163 [258] | 543 [288] | 24 [1,373] |

### 5.2 Comparison with Other PGAS Implementations

In this section, we present the performances and productivities of the HPCC benchmarks using other PGAS languages. Table 1 shows the performances and the SLOC of RandomAccess, HPL, and FFT at the HPCC Awards class 2 in 2011 and 2012, and the XMP implementations. Note that not only the SLOC but also the performance is only one of indicators of performance. This is because the performance of each benchmark is strongly related to the characteristics of the machine.

Table 1 shows that the SLOC of HPL written in XMP are relatively good. The reason is that the **gmove** directive of XMP automatically generates complicated communication and pack/unpack operations. The SLOC of FFT written in XMP is a large number because we have implemented only the main kernel of all FFT subroutines. The other kernels are almost the same as the hpcc-1.4.2 FFT written in MPI. The performance of RandomAccess written in XMP is much worse than that written in X10. The reason is that IBM Power 775 has a significant high-speed interconnect (its peak bi-directional interconnect bandwidth of 192 GB/s)[27].

## 6 Conclusion and Future Research

The present paper describes the proposed implementations of RandomAccess, HPL, and FFT of the HPCC benchmarks using XMP. Through these implementations, we demonstrate that the XMP programming model has good productivity for an HPC system. Moreover, we evaluated the performances on the K computer, which is a leadership HPC system. The results using a maximum of 262,144 CPU cores show that XMP has scalable performances. There are two important reasons for the good performance. First, XMP coarray syntax directly uses a high-speed extended RDMA interface on the K computer. Second, XMP can tune the application while being aware of local memory because XMP provides functions to obtain local memory information for a global array. These results show that XMP has good strategies for high performance and productivity of HPC applications.

In the future, several investigations have been described as follows. First, we will rewrite part of the code on the local-view model in order to increase the performance. Second, the internal packing operation in the **gmove** directive will support thread parallelization. Third, we will develop real applications using XMP. Finally, we will implement applications using XMP and OpenACC on an accelerator cluster and evaluate their productivity and performance.

## Acknowledgments


- Modification of the two functions for sorting and updating of the local table in RandomAccess were performed in cooperation with Fujitsu Limited.

- The authors would like to thank Ikuo Miyoshi who belongs to Fujitsu Limited for his lecture on HPL.

- The present 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] `http://www.netlib.org/blas/`.

[2] `http://www.netlib.org/scalapack/`.

[3] Ffte: A fast fourier transform package. `http://www.ffte.jp`.

[4] Ha-pacs. `http://www.ccs.tsukuba.ac.jp/CCS/eng/research-activities/projects/ha-pacs`.

[5] Hpc challenge benchmarks. http://icl.cs.utk.edu/hpcc/.

[6] Omni xcalablemp compiler. http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/xcalablemp/.

[7] Openacc home. http://www.openacc-standard.org.

[8] Pc cluster consortium. http://www.pccluster.org/en/.

[9] Top500 supercomputer sites. http://www.top500.org.

[10] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In The 20th Int'l Parallel and Distributed Processing Symposium (IPDPS), 2006.

[11] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[12] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 2:1–2:3, New York, NY, USA, 2010. ACM.

[13] UPC Consortium. Upc language specifications. Technical Report LBNL-59208, Berkeley National Laboratory, 2005. http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf.

[14] Frontiers of Massively Parallel Computation. *Communication overhead on CM5: an Experimental Performance Evaluation*, 1992.

[15] FUJITSU LIMITED. *Parallelnavi Technical Computing Language MPI User's Guide*, 2013.

[16] Jorge Gonzlez-Domnguez, Mara J. Martn, Guillermo L. Taboada, Juan Tourio, Ramn Doallo, Damin A. Malln, and Brian Wibecan. Upcblas: a library for parallel matrix computations in unified parallel c. *Concurr. Comput. : Pract. Exper.*, 24(14):1645–1667, September 2012.

[17] Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of high performance fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 7–1–7–22, New York, NY, USA, 2007. ACM.

[18] Charles H. Koelbel, David B. Loverman, Robert S. Shreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[19] Jinpil Lee. *A Study on Productive and Reliable Programming Environment for Distributed Memory System*. PhD thesis, University of Tsukuba, 2012.

[20] Jinpil Lee, Minh Tuan Tran, Tetsuya Odajima, Taisuke Boku, and Mitsuhisa Sato. An extension of xcalablemp pgas lanaguage for multi-node gpu clusters. In *Proceedings of the 2011 international conference on Parallel Processing*, Euro-Par'11, pages 429–439, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] Nakao Masahiro, Lee Jinpil, Boku Taisuke, and Sato Mitsuhisa. Xcalablemp implementation and performance of nas parallel benchmarks. In *Proceedings of the Fourth Conference on Partitioned*

*Global Address Space Programming Model*, PGAS '10, pages 11:1–11:10, New York, NY, USA, 2010. ACM.

[22] Nakao Masahiro, Lee Jinpil, Boku Taisuke, and Sato Mitsuhisa. Productivity and performance of global-view programming with xcalablemp pgas language. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, CCGRID '12, pages 402–409, Washington, DC, USA, 2012. IEEE Computer Society.

[23] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global arrays: A non-uniform-memory-access programming model for high-performance computers. *THE JOURNAL OF SUPERCOMPUTING*, 10:169–189, 1996.

[24] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[25] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification, 2013. http://x10.sourceforge.net/documentation/languagespec/x10-231.pdf.

[26] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[27] Olivier Tardieu, David Grove, Bard Bloom, David Cunningham, Benjamin Herta, Prabhanjan Kambadur, Vijay A. Saraswat, Avraham Shinnar, Mikio Takeuchi, and Mandana Vaziri. X10 for productivity and performance at scale. RC25334 (WAT1210-081), October 2012.

[28] XcalableMP Specification Working Group. Xcalablemp specification version 1.1, 11 2012. `http://www.xcalablemp.org/spec/xmp-spec-1.1.pdf`.

[29] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.