Performance Evaluation for Omni XcalableMP Compiler on Many-core Cluster System based on Knights Landing

Masahiro Nakao  
RIKEN Advanced Institute for Computational Science  
Hyogo, Japan  
masahiro.nakao@riken.jp

Hitoshi Murai  
RIKEN Advanced Institute for Computational Science  
Hyogo, Japan

Taisuke Boku  
Center for Computational Sciences  
University of Tsukuba  
Ibaraki, Japan

Mitsuhisa Sato  
RIKEN Advanced Institute for Computational Science  
Hyogo, Japan

ABSTRACT
To reduce the programming cost on a cluster system, Partitioned Global Address Space (PGAS) languages are used. We have designed an XcalableMP (XMP) PGAS language and developed the Omni XMP compiler (Omni compiler) for XMP. In the present study, we evaluated the performance of the Omni compiler on Oakforest-PACS, which is a cluster system based on Knights Landing, and on a general Linux cluster system. We performed performance tuning for the Omni compiler using a Lattice QCD mini-application and some mathematical functions appearing in that application. As a result, the performance of the Omni compiler after tuning was improved compared to before tuning on both systems. Furthermore, we compared the performance of MPI and OpenMP (MPI+OpenMP), which is an existing programming model, to that of XMP with the tuned Omni compiler. The results showed that the performance of the Lattice QCD mini-application using XMP was achieving more than 94% of the implementation written in MPI + OpenMP.

CCS CONCEPTS
• Software and its engineering → Parallel programming languages;

KEYWORDS
Knights Landing, Cluster system, PGAS language, Compiler

1 BACKGROUND
For developing parallel applications on cluster systems, Partitioned Global Address Space (PGAS) languages [1, 3, 7, 10] are used to reduce the programming cost. A PGAS language creates a global memory space from a unique memory space of each machine and provides users with a global address to access the created global memory space. By using a PGAS language, a user can program like a shared memory system in a cluster system. This feature enables users to develop parallel applications with ease.

We have designed an XcalableMP (XMP) PGAS language [4, 8, 9], which is a directive-based language extension based on the C and Fortran languages. XMP also specifies that some OpenMP directives can be combined into XMP directives for thread programming. In addition, we have developed the Omni XMP compiler (Omni compiler) [11] as a compiler system for XMP. The Omni compiler is a source-to-source compiler that translates code written in XMP into parallel code. In previous studies [8, 9], we evaluated the performance of XMP and the Omni compiler on the K computer [5] and on general Linux cluster systems. However, we have little experience with XMP on cluster systems based on many-core processors which have been attracting attention in the HPC field.

In this study, we evaluate the performance of the Omni compiler on Oakforest-PACS [6], which is a cluster system based on Knights Landing, and a general Linux cluster system. Moreover, to evaluate the performance of XMP, we implement a Lattice QCD mini-application, which is an important application in the HPC field. Specifically, this study makes the following key contributions: (1) We describe how to implement the Lattice QCD mini-application using a hybrid model of XMP and OpenMP. (2) We describe an effective code translation method for a source-to-source compiler.

The remainder of this paper is structured as follows. Sections 2 and 3 give overviews of XMP and the Omni compiler, respectively. Section 4 discusses the performance tuning of the Omni compiler on Knights Landing and a general CPU. Section 5 evaluates the performance of the Lattice QCD mini-application and of some mathematical functions appearing in this application. Section 6 summarizes this paper and discusses areas for future research.
2 XCALABLEMP

The XMP specification has been designed by the PC Cluster Consortium (https://www.pccluster.org/en/). XMP provides directives for data mapping, work mapping, and communication to develop a parallel application for cluster systems. This section describes the XMP C language syntax required for implementing the Lattice QCD mini-application.

2.1 Data Mapping

XMP can declare a distributed array on the global memory space. In order to express the distribution method, a virtual index set called a “template” is used. The template, node, distribute, align directives are used to declare the distributed array.

Fig. 1 shows an XMP example code for declaring a distributed array. In Fig. 1a, the template directive defines a template t that has index values from 0 to N-1. An XMP execution unit is called a “node,” and each node executes redundantly in parallel. The node directive defines a node set p that has four nodes. This means an application is executed by four nodes. If “*” is used in place of a number in square bracket (as in p[*]), then the number of nodes is dynamically determined when a program runs. The distribute directive distributes t onto p in a block manner, meaning that the elements of template t are distributed with block sizes as equal as possible. XMP also provides cyclic, block-cyclic, and uneven-block distributions. The align directive aligns the distributed array a[i] with t. In the case of N = 16, each node has four elements of a[j]. XMP also declares a multi-dimensional distributed array. Fig. 1b shows how to declare a two-dimensional distributed array b[i][j] using a two-dimensional template and a two-dimensional node set. Please refer to [10] for the details of the multi-dimensional distribution.

2.2 Work Mapping

The loop directive parallelizes the next loop statement depending on a template. Fig. 2 shows an XMP example code where the distributed array a[i] and template t defined in Fig. 1a are used.

In Fig. 2a, the loop directive parallelizes a loop statement depending on template t. In the case of N = 16, a node p[0] executes index i from 0 to 3. Fig. 2b shows an example of a hybrid program using XMP and OpenMP directives. In Fig. 2b, firstly, the loop directive parallelizes a loop statement across nodes, and then the parallel for directive also parallelizes the loop statement parallelized by the loop directive across threads within a node. The order of the loop directive and the parallel for directive does not matter in this case.

2.3 Communication

2.3.1 Reduction. To support an addition expression in a loop statement, a reduction clause can be added to the loop directive. Fig. 3 shows an XMP example code. In Fig. 3a, a reduction clause is added to the loop directive. The reduction clause performs a reduction operation for the local variable sum located in each node when finishing the loop statement. In Fig. 3b, the reduction clauses are added to the loop and parallel for directives in order to perform a reduction operation for thread programming.
2.3.2 Halo exchange. For easy development of stencil applications, XMP provides shadow and reflect directives. Fig. 4 shows the concepts of these directives where a distributed array $a[i]$ is the array declared in Fig. 1a and $N = 16$. In Fig. 4a, the shadow directive adds halo regions to both sides of a distributed array $a[i]$. The halo regions appear as gray cells. The width of the halo region is 1, which is indicated in the shadow directive. In Fig. 4b, the reflect directive synchronizes the halo regions between neighboring nodes. Note that global boundary (the left halo of $p[0]$ and the right halo of $p[3]$) is not updated in the example. When the global boundary is also updated, the periodic clause should be added to the reflect directive.[10]

3 OMNI XCALABLEMP COMPILER

3.1 Overview

Fig. 5 shows the compile flow of the Omni compiler. Firstly, the Omni compiler translates XMP directives in user code into XMP runtime calls. If necessary, code besides the XMP directives is also modified. Secondly, a native compiler (e.g., gcc, Intel, or PGI) compiles the translated code and creates an execution binary with linking to an XMP runtime library.

3.2 Example of Code Translation

This section describes how to translate user code by using Omni compiler-1.2.2, which is the latest stable version1. The distributed array $b[i][j]$ and template $t$ used in this section are defined in Fig. 1b and $N = M = 10$.

1http://omni-compiler.org/download/stable/omnicompiler-1.2.2.tar.bz2

HPC Asia 2018 WS, January 31, 2018, Chiyoda, Tokyo, Japan

3.2.1 Distributed array. Fig. 6 shows an XMP example code of an align directive and an array. The align directive deletes a declaration of an array $b[i][j]$ in the user code and creates a pointer _XMP_ADDR_b and a descriptor _XMP_DESC_b for a new array in the translated code. Moreover, the align directive adds some functions to allocate memory for the new array. 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. (For example, “*” is used in the node set described in section 2.1)

3.2.2 Loop statement. Fig. 7 shows an XMP example code of an loop directive and a loop statement. In Fig. 7a, the loop directive parallelizes the next nested loop statement depending on the two-dimensional template $t$. In Fig. 7b, the initial value ($i = 0, j = 0$) and ending condition ($i < 5, j < 5$) in the loop statements are described as constants automatically in this case. If an iteration differs for each node, then the iteration is calculated just before the loop statement by an XMP function and the results are used in its initial value and ending condition as variables.

4 IMPLEMENTATION OF LATTICE QCD

4.1 Overview of Lattice QCD

Quantum chromodynamics (QCD) is a fundamental equation representing a quark, which is a species of elementary particle, and a gluon, which is the particle that mediates the strong interaction. The Lattice QCD is a discrete formulation of QCD that simulates in a lattice of four dimensions (time: T and space: ZXY). The quark degrees of freedom are represented as a field that has four components: spin and three color components. The gluon is defined as a...
Figure 8: Lattice QCD pseudo-code

```c
typedef struct Quark {
  double v[4][3][2];
} Quark;

typedef struct Gluon {
  double v[3][3][2];
} Gluon;

type[3][2];

#pragma xmp template t[NT][NZ] width(0,0,0,0)
#pragma xmp reflect(u) width(0,0,0,0)
#pragma xmp loop (t,z) on t[0][0]
#pragma xmp loop (t,z) on t[0][0]
#pragma xmp align u[0][1][1][0][0]
#pragma xmp align v[1][1][0][0]
#pragma xmp shadow tmp_v[1][1][0][0]
#pragma xmp shadow tmp_v[1][1][0][0]
#pragma xmp shadow tmp_v[1][1][0][0]
#pragma xmp shadow tmp_v[1][1][0][0]
```

Figure 9: Declaration of distributed arrays for Lattice QCD

A 3 × 3 complex matrix. In a Lattice QCD simulation, one needs to solve many times a linear equation for the matrix that represents the interaction between the quark and gluon fields.

4.2 Implementation in XcalableMP

We implemented a Lattice QCD mini-application based on the existing Lattice QCD application Bridge++ [2]. Our code extracted the main function of Bridge++.

Fig. 8 shows pseudo-code of our implementation where \( U \) is a gluon, the other uppercase characters are quarks, and lowercase characters are scalar variables. The WD() in lines 5, 6, 11, and 12 is the Wilson–Dirac operator [12], which is a main kernel in the Lattice QCD mini-application for calculating the interactions of the quarks under the influence of the gluon. Our implementation uses the CG method to solve the linear equation. For the CG method, the following mathematical functions for the quark matrix are implemented. (1) COPY method in lines 1–3 and 8, which copies to the left-hand side from the right-hand side. (2) NORM method in lines 4, 9, and 17 calculates the L2-norm. (3) AXPY method in lines 7, 15, 16, and 20 adds matrices. (4) DOT method in line 13 calculates a dot product. (5) SCAL method in line 19 multiplies by a scalar. Lines 14, 18, and 21 are scalar-to-scalar operations.

Fig. 9 shows the declaration of the distributed arrays of the quark and gluon. Specifically, in lines 1–8, the quark and gluon structure arrays are declared. The last dimension "[2]" of each structure represents the real and imaginary parts of a complex number.

Macros NT, NZ, NY, and NX are the numbers of TXYZ axis elements. Macros PT and PZ in the nodes directive of line 11 are the numbers of nodes on only the T and Z axes. Thus, the program is parallelized on the T and Z axes. In the align directives of lines 13–15, an "=" in square brackets means that the dimension is not divided. In the shadow directive of lines 16–18, a "0" in square brackets means that no halo region exists.

Fig. 10 shows a portion of the Wilson–Dirac operator code. In line 3, the loop directive parallelizes the outer two loop statements. In line 4, the parallel for directive with the collapse clause also parallelizes all loop statements parallelized by the loop directive. In the loop statements, one calculation needs neighboring elements of distributed arrays. Fig. 11 shows calling WD() where the reflect directives are used before WD() to synchronize own halo region between neighboring nodes. The width and orthogonal clauses restrict the transfer range of the halo region to reduce communication time [4, 10]. The halo region of \( u \) is not synchronized before the second WD() because values of \( u \) are not updated in WD().

Fig. 12 shows SCAL method where loop statements comprise seven nested loops, and are parallelized using XMP and OpenMP directives. COPY, NORM, AXPY, and DOT methods are shown...
Performance Evaluation for Omni XcalableMP Compiler on Many-core Cluster System based on Knights Landing

In Figs. 19 – 22 in Appendix. The reason for specifying `collapse(4)` instead of `collapse(7)` in the `parallel for` directive is because the performance of `collapse(4)` was better than that of `collapse(7)` in our preliminary evaluation.

5 PERFORMANCE EVALUATION

5.1 Tuning Omni XcalableMP Compiler

In our preliminary evaluation using Omni compiler-1.2.2, the performance of the mathematical functions deteriorated. Fig. 13 shows a portion of translated SCAL code from Fig. 12 generated by Omni compiler-1.2.2. In Fig. 13, NT, NZ, NY, and NX are 32, and PT and PZ are 2. In line 9, the reason why “1” is added to t and z is because it calculates the size of a halo. Here, the “2” in the (16 + 2) term indicates the size of the halo of both sides. Because the distributed array v is one dimensional, it is difficult for a native compiler to effectively optimize the code.

In order to solve this problem, the code conversion of Omni compiler-1.2.2 has been changed to leave the size information of each dimension just before a target loop statement if possible (For example, when node set is determined dynamically, the new code conversion is not performed). Fig. 14 shows a portion of translated SCAL code from Fig. 12 generated by the new Omni compiler. In line 1, a new pointer `_XMP_MULTI_ADDR_v` is declared which has the size of each dimension. The new pointer is defined as a head pointer of the distributed array v. In a loop statement, operations are performed using the new pointer.

5.2 Performance of Mathematical Functions

In order to investigate the influence of the code conversion change described in section 5.1, we conduct performance evaluations of the mathematical functions in Figs. 12, 19 – 22. We use a single compute node of not only Oakforest-PACS but also the COMA system (COMA) as a general Linux cluster system. Tables 1 and 2 show the evaluation environments. We use two program sizes: (NT, NZ, NY, NX) = (32, 32, 32, 32) and (2, 32, 32, 32). The compile options on Oakforest-PACS are set as “-O2 -mcmodel=medium -axMIC-AVX512”, whereas those on COMA are set as “-O2 -mcmodel=medium”. The number of processes is 1 on both machines, but the numbers of threads are 64 on Oakforest-PACS and 10 on COMA. Memory mode of Oakforest-PACS is “cache” which means MCDRAM works for DDR4 memory as a cache. We use the Intel compiler as the backend compiler of the Omni compiler. We execute the mathematical functions 100 times and measure the elapsed time.
Figure 16: Performance of mathematical functions on COMA

(a) (NT, NZ, NY, NX) = (32, 32, 32, 32)

(b) (NT, NZ, NY, NX) = (2, 2, 32, 32)

Figure 17: SCAL operator code using OpenMP

5.3 Performance of Lattice QCD

This section evaluates the performance of the Lattice QCD mini-application written in XMP using Omni compiler-1.2.2 and the new Omni compiler. For comparison purposes, an implementation of the application written in a combination of MPI and OpenMP (MPI + OpenMP) is also evaluated. While we assigned one process per compute node on Oakforest-PACS, we assigned two processes per computer node on COMA because the computer node of COMA has two CPU sockets. We performed up to 256 processes on both systems. The compile options, the number of threads, the memory mode, and the backend compiler are the same as those in the section 5.2. We execute the Lattice QCD code with strong scaling in regions (32, 32, 32, 32) as (NT, NZ, NY, NX).

Fig. 18 shows the performance results. All the performance results using the new Omni compiler are better than those using Omni compiler-1.2.2. In addition, XMP using the new Omni compiler achieves a performance of more than 94% of MPI + OpenMP. Examining the reason why the performance of XMP is a little worse than MPI in 256 processes, as shown in Figs. 15b and 16b, the performance of the mathematical functions except for COPY using the new Omni compiler is the almost same or a little worse than that of only OpenMP. Although the performance of COPY using the new Omni compiler is better, the function COPY is only called a small number of times, as shown in Fig. 8.

6 SUMMARY AND FUTURE WORK

In this paper, we examined the tuned performance of the Omni compiler for an XMP PGAS language on a Knights Landing cluster system and a general Linux cluster system. Specifically, we evaluated the XMP performance through an implementation of a
Lattice QCD mini-application. As a result of evaluating the Omni compiler before and after the performance tuning, we found that the performance of the Omni Compiler after tuning was superior. The results also showed that the performance of the Lattice QCD mini-application written in XMP using the tuned compiler is achieving more than 94% of the implementation written in MPI+OpenMP. Future research will examine the performance difference between XMP and MPI+OpenMP more deeply and also evaluate the performance of other applications besides the Lattice QCD mini-application.

ACKNOWLEDGEMENTS

We would like to extend grateful thanks to Hideo Matsufuru who provided us the Lattice QCD code. This research used the Oakforest-PACS and COMA systems provided by Interdisciplinary Computational Science Program in the Center for Computational Sciences, University of Tsukuba. The work was supported by the Japan Science and Technology Agency, Core Research for Evolutional Science and Technology program entitled “Research and Development on Unified Environment of Accelerated Computing and Interconnection for Post-Petascale Era” in the research area of “Development of System Software Technologies for Post-Peta Scale High Performance Computing.”

A MATHMATICAL FUNCTIONS

Performance Evaluation for Omni XcalableMP Compiler on Many-core Cluster System based on Knights Landing

REFERENCES