XcalableMP 2.0 and Future Directions

Mitsuhisa Sato, Hitoshi Murai, Masahiro Nakao, Keisuke Tsugane, Tesuya Odajima, and Jinpil Lee

Abstract This chapter presents the XcalableMP on the Fugaku supercomputer, the Japanese flagship supercomputer developed by FLAGSHIP2020 project in RIKEN R-CCS. The porting and the performance evaluation were done as a part of this project, and the XcalableMP is available for the Fugaku users for improving the productivity and performance of parallel programing. The performance of XcalableMP on the Fugaku is enhanced by the manycore processor and a new Tofu-D interconnect. We are now working on the next version, XcalableMP 2.0, for cutting-edge high-performance systems with manycore processors by multithreading and multi-tasking with integrations of PGAS model and synchronization models. We conclude this book with retrospectives and challenges for future PGAS models.

1 Introduction

We have been developing a production-level XcalableMP compiler, and make it available for the K computer's users as well as the users of conventional clusters. RIKEN R-CCS has been carrying out the FLAGSHIP 2020 Project [1] to develop the Japanese flagship supercomputer system following the K computer, the Post-K, formally named as "Fugaku" later, since 2014. In the project, XcalableMP was taken as a parallel programming language project for improving the productivity and performance of parallel programing. XcalableMP is now available on Fugaku and the performance is enhanced by the Fugaku interconnect, Tofu-D. The next section describes the XcalableMP on Fugaku.

M. Sato (✉) · H. Murai · M. Nakao · T. Odajima · J. Lee
RIKEN Center for Computational Science, Kobe, Japan
e-mail: msato@riken.jp; h-murai@riken.jp; masahiro.nakao@riken.jp; tetsuya.odajima@riken.jp; jinpil.lee@riken.jp

K. Tsugane
Fujitsu Laboratories Ltd., Kawasaki, Kanagawa, Japan
e-mail: tsugane.keisuke@fujitsu.com

M. Sato (ed.), *XcalableMP PGAS Programming Language*,
https://doi.org/10.1007/978-981-15-7683-6_10

The XcalableMP project has been started from 2008 and the discussion on XcalableMP 1.x has converged. We are now working on a new version, XcalableMP 2.0, targeted for cutting-edge high-performance systems with manycore processors by multithreading and multi-tasking with integrations of PGAS model and synchronization models. In this new programming model, the execution of the program is decomposed into several tasks executed according the dependency between tasks. This model will enable less overhead of synchronization by eliminating expensive global synchronization, overlap between computation and communication in many-core, and light-weight communication by RDMA in PGAS model. We will extend this programming model to combine several kinds of accelerators such as GPU, FPGA, and special-purpose processors with large-scale general-purpose manycore systems. It enables some tasks to be offloaded into the accelerators such as FPGA as well as each core in modern manycore processor. We consider this configuration as a general global architecture of the future system as some part of system will be specialized for high performance and power efficiency. Our programming model will make it easy to adopt the existing computational science program to the new systems.

In Sect. 3, a proposal for XcalableMP 2.0 is presented, followed by retrospectives and challenges for future PGAS models in Sect. 4.

2 XcalableMP on Fugaku

In this section, we report our early experience and the preliminary performance of XcalableMP on Fugaku. The Fugaku is a huge-scale system with general-purpose manycore processors. The node processor is a single chip, Fujitsu A64FX, which consists of 48 cores with 2 or 4 cores dedicated for OS activities, 32 GiB HBM2 memory, with Tofu-D interconnect, and a PCI express controller in the chip together. The Fugaku system consists of 158,976 nodes in 432 racks. The Fugaku is scheduled to be put into operation for public service around 2021. In 2020, the installation is completed and the system partially serves the early access program.

XcalableMP is available as a parallel programming language for the Fugaku, supported by R-CCS team with Fujitsu. C and Fortran are supported as base languages with XcalableMP 1.2 compliant.

We report the preliminary performance of XcalableMP program running on the Fugaku.[1]

We used the following versions:

- Omni XcalableMP Version: 1.3.2, Git Hash: 6d23f46.
- Language specification: 1.2.25.

[1]The reported results were obtained on the evaluation environment in the trial phase. Note that the performance is not guaranteed at the start of its operation.

The performance of XcalableMP on the Fugaku is enhanced by the manycore processor and a new Tofu-D interconnect.

2.1 Performance of XcalableMP Global View Programming

We executed the IMPACT-3D, described in Chap. 6, for the evaluation of XcalableMP global view programming in the Fugaku, using up to 512 nodes. The scalability on Fugaku is shown in Fig. 1, comparing to the MPI version. The program is parallelized by hybrid XMP-OpenMP parallel programming: An XMP node is assigned to a node, and 48 OpenMP threads are running within a node. The problem size is $512 \times 512 \times 512$ with three-dimensional block distribution. The compile option is "-Kfast".

As shown in the figure, we found a good scalability in Fugaku, and the performance is better than that by MPI thanks to the optimized XMP runtime for communications in the stencil computation [2].

2.2 Performance of XcalableMP Local View Programming

Fugaku has a customized interconnection, called Tofu-D, which provides hardware-supported RDMA (Remote Direct Memory Access) operations. We implemented the XMP runtime library to make use of Tofu-D for one-sided communication for

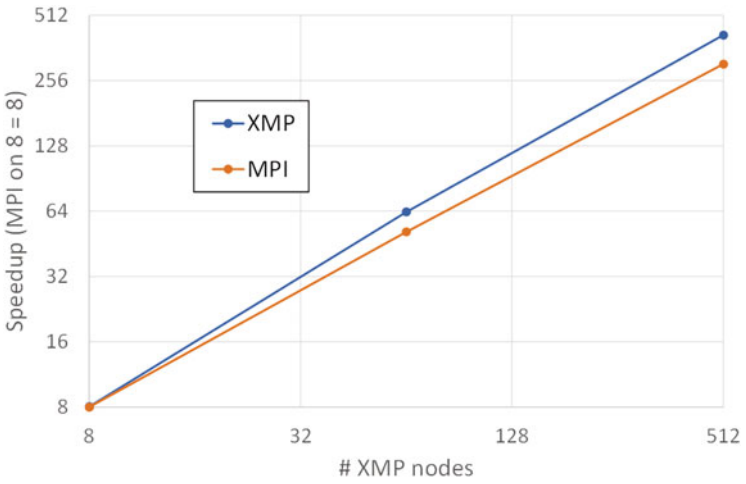

Fig. 1 Speedup of Impact3D on Fugaku and performance comparing to K computer

the XMP local view programming. The library is implemented by using a low-level communication layer, uTofu API [3], provided by Fujitsu.

For performance evaluation of XMP local view programming, we used CCS QCD and NTChem-MINI taken from the coarray version of Fiber Miniapp Suite [4, 5].

To run CCS QCD mini-application [6], eight XMP nodes are assigned to one node, running in a flat XMP mode. The size and conditions are as follows:

- Target data: Class 2 ($32 \times 32 \times 32 \times 32$) (strong scaling).
- Compiler options: -Kfast, zfill, simd=2.
- Timing region: sum of "Clover + Clover_inv Performance" and "BiCGStab (CPU: double precision) Performance" of the built-in timing feature.

Figure 2 shows the speedup of the Fugaku, comparing to the performance of the K computer. The XMP version archives almost same performance of the MPI version. Note that the reason of the performance degradation of the XMP version on the K computer is the overhead of allocation for allocatable coarray used as a buffer for communication. It is improved by removing this overhead by using the uTofu communication layer.

The NTChem-MINI is a mini-application taken from NTChem [7], a high-performance software package for molecular electronic structure calculation. An XMP node is assigned to one node, and within a node, BLAS functions are executed using 48 cores. The size and conditions are set as follows:

- Target data: taxol (strong scaling).
- Compiler options: -Kfast, simd=2.
- Timing region: "RIMP2_Driver" of the built-in timing feature.

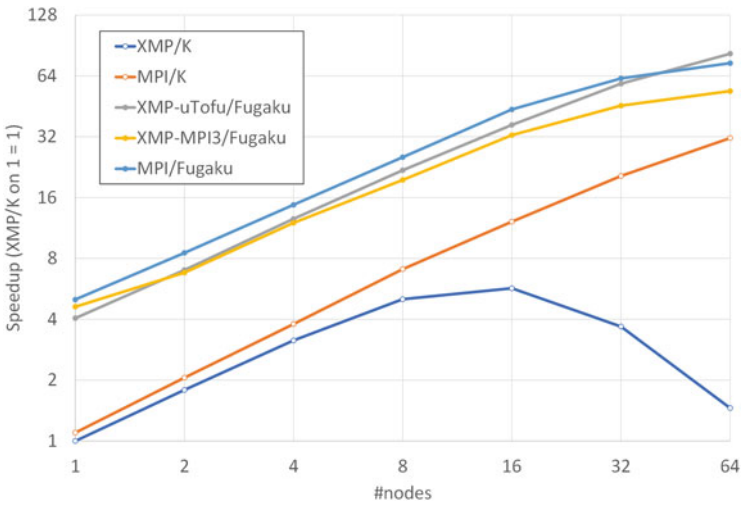

Fig. 2 Speedup of CCS QCD on Fugaku and performance comparing to the K computer

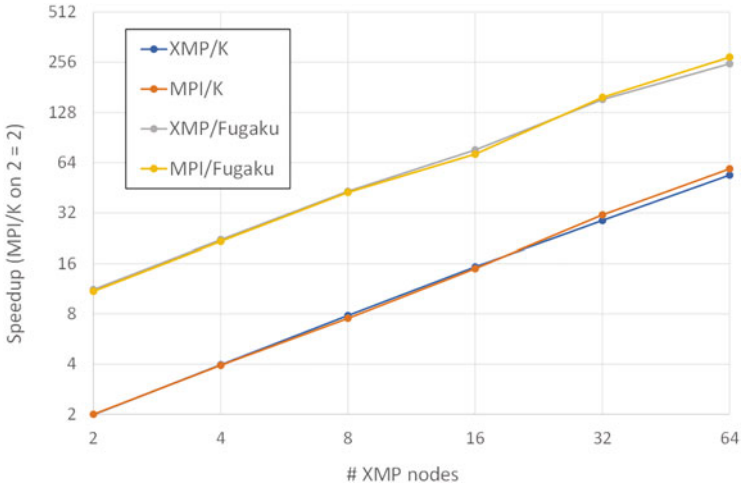

Fig. 3 Speedup of NTChem-MINI on Fugaku and performance comparing to the K computer

As shown in Fig. 3, the XMP versions archive almost the same performance of the original MPI versions.

3 Global Task Parallel Programming

Recently, large-scale clusters of manycore processors such as Intel Xeon Phi have been deployed in many sites from the latest Top500 Lists. In order to program manycore processors, OpenMP is widely used as a shared-memory programming model. Most OpenMP programs are written using work sharing constructs for loops, which involves a global synchronization. However, especially in modern manycore processors, the global synchronization cost for work sharing becomes bigger, and the load imbalance among cores lead to the performance degradation as the number of cores on the processor increases. Task parallel programming using task dependency in OpenMP 4.0 is a promising candidate to facilitate the parallelization for such manycore processors because it enables users to avoid global synchronization by fine-grained task-to-task synchronization through user-specified data dependencies.

We are interested in extending the task parallel programming model to the PGAS model of XcalableMP for distributed memory systems. As well as removing expensive global synchronization, it is expected to enable the overlapping of communication and computation. For XMP 2.0, we propose the global task parallel programming.

In OpenMP, the task dependency in a node depends on the order of reading and writing to data based on the sequential execution. Therefore, the OpenMP multi-

tasking model cannot be applied to describe the dependency between tasks running in different nodes since threads of each nodes are running in parallel.

We propose new directives for communication with tasks in XMP, and they enable users to write easily the multi-tasking execution based on XMP language constructs. The tasklet directive generates a task for the associated structured block on the node specified by the on clause, and the task is scheduled and immediately executed by an arbitrary thread in the specified node if there is no task dependency. If it has any task dependencies, the task execution is postponed until all dependencies are resolved. The tasklet gmove directive copies the variable of the right-hand side (RHS) into the left-hand side (LHS) of the associated assignment statement for local or distributed data in tasks. If the variable of the RHS or the LHS is the remote data, this directive may synchronize on data dependency between nodes and execute communication. The tasklet reflect directive is a task-version of reflect operation. It updates halo regions of the array specified to array-name in tasks. In this directive, data dependency is automatically added to these tasks based on the communication data because the boundary index of the distributed data is dynamically determined by XMP runtime system.

We have designed a simple code translation algorithm from the proposed directives to XMP runtime calls with MPI and OpenMP. We have evaluated the performance using block-Cholesky Factorization Program on KNL based-system, Oakforest-PACS. Through the experiment, we confirmed the advantage of task parallelism over the traditional loop-based data parallelism. At the same time, we found the performance problems on communication between multiple threads (MPI_THREAD_MULTIPLE). Currently, we are investigating a lower-level communication API for efficient one-sided communication of PGAS operations in multithreaded execution environment.

Details of the proposal in this chapter are described in [8].

3.1 OpenMP and XMP Tasklet Directive

While OpenMP originally focuses on work sharing for loops as the `parallel for` directive, OpenMP 3.0 introduces task parallelism using the `task` directive. It facilitates the parallelization where work is generated dynamically and irregularly as in recursive structures or unbounded loops. The `depend` clause on the `task` directive is supported from OpenMP 4.0 and specifies data dependencies with dependence-type `in`, `out`, and `inout`. Task dependency can reduce the global synchronization of a thread team because it can execute fine-grained synchronization between tasks through user-specified data dependencies.

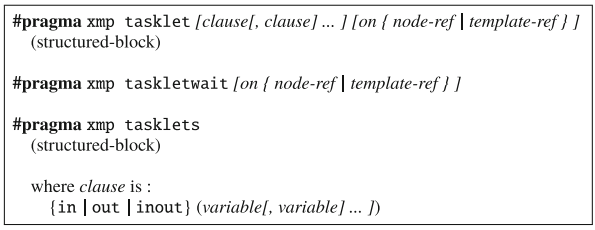

Fig. 4 Syntax of the `tasklet`, `taskletwait`, and `tasklets` directives in XMP

To support task parallelism in XMP as in OpenMP, the `tasklet` directive[2] is proposed in XMP 2.0. Figure 4 describes the syntax of the `tasklet`, `tasklets`, and `taskletwait` directives for the multi-tasking execution in XMP. The `tasklet` directive generates a task for the associated structured block on the node specified by the `on` clause, and the task is scheduled and immediately executed by an arbitrary thread in the specified node if there is no task dependency. If it has any task dependencies, the task execution is postponed until all dependencies are resolved. These behaviors occur when these tasks are surrounded by `tasklets` directive. When these tasks are not surrounded by the `tasklets` directives, they are executed sequentially at the specified node. The `tasklet` directive supports several clauses for the description of the task dependency. The `in`, `out`, and `inout` clauses represent the task dependency in a node. When `in`, `out`, or `inout` clause presents on the `tasklet` directive, the generated task has each data dependency in a node. The behavior of these data dependencies is same as OpenMP `task depend` clause: flow, anti, and output dependencies.

The `taskletwait` directive waits on the completion of the generated tasks on each node. Since the directive does not involve the barrier synchronization, the `barrier` directive in XMP is also required in order to guarantee that all tasks of all nodes are finished at this point. There is an implicit barrier on each node at the end of the `tasklets` directive.

In OpenMP, the task dependencies are created according to the order of reading and writing to data based on the sequential execution in a node. Therefore, the OpenMP task parallel model cannot be directly applied to describe the dependency between tasks running in different nodes since threads of each nodes are running in parallel.

In OmpSs [10], interactions between nodes are described through the MPI task that is executing MPI communications. Task dependency between nodes is guaranteed by the completion of MPI point-to-point communication in tasks. While this approach can satisfy dependencies between nodes, it may cause further productivity degradation because it forces users to use a combination of two programming models that are based on different description formats. Therefore, we propose new directives for communication with tasks in XMP, and they enable

[2]There is the `task` directive in XMP, it is different from OpenMP's one.

users to write easily the multi-tasking execution for clusters by only using language constructs.

3.2 A Proposal for Global Task Parallel Programming

In order to support multi-tasking execution for distributed memory parallel systems, we need to perform point-to-point communication within tasks in local task dependency graphs. While XMP provides some directives for communication, many of these are performed collectively, and cause an implicit synchronization among execution nodes. This causes a performance degradation, because tasks participating in communications, such as broadcast, wait for synchronization until all tasks are completed. For XMP 2.0, we propose two directives, `tasklet gmove` and `tasklet reflect`, as shown in Fig. 5, to describe interactions between nodes in tasks by point-to-point communication, for inter-node data dependency. These communications are only synchronized between the sender and receiver of the communication in each task.

These details are as follow:

`tasklet gmove` directive: Although this copies the variable from the right-hand side (RHS) into the left-hand side (LHS) of the associated assignment statement for local or distributed data like the `gmove` directive, it is executed in tasks. The copy operation is basically performed on all execution nodes. However, if the distributed array is specified at the associated assignment statement, only nodes with the distributed array execute the operation in the task. The execution nodes can also be determined by the `on` clause. When the `in`, `out`, or `inout` clause is present on the `tasklet gmove` directive, the generated task has the corresponding data dependency in a node, similar to the `tasklet` directive.

`tasklet reflect` directive: Although this updates halo regions of the array specified to `array-name` as in the `reflect` directive, it is executed in tasks. For example, when updating one side of a halo region for a one-dimensional distributed array on two nodes, these communications are separated into four tasks: the sender of the upper element on node 1, the receiver of the upper halo region on node 1, the sender of the lower element on node 2, and the receiver of the lower halo region on node 2. In this directive, data dependency is automatically added to these generated tasks based on the communication data, because the boundary index of the distributed array is dynamically determined by the XMP

Fig. 5 Syntax of the `tasklet gmove` and `tasklet reflect` directives in XMP

#pragma `xmp tasklet gmove` *[clause[, clause] ...] [on { node-ref | template-ref }]*
(an assignment statement)

#pragma `xmp tasklet reflect` *(array-name[, array-name] ...)*
[blocksize (reflect-blocksize[, reflect-blocksize] ...)]

where *clause* is :
{`in` | `out` | `inout`} (*variable[, variable] ...]*)

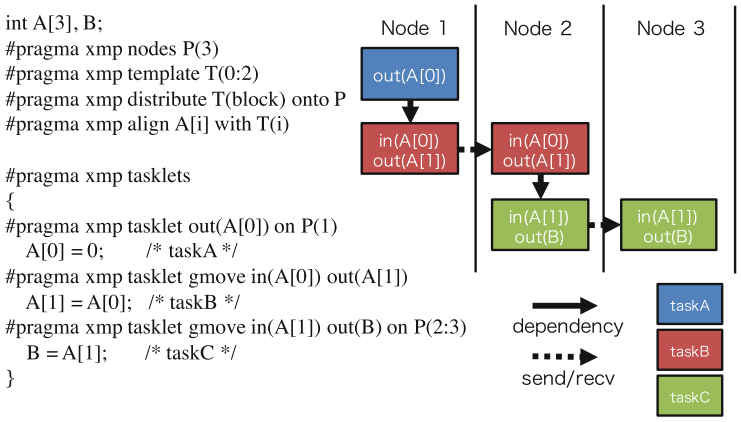

Fig. 6 Example of the `tasklet` and `tasklet gmove` directives

runtime system. The `chunksize` clause can be matched the task dependency descriptions of users using the dependency generated by the `tasklet reflect` directive. When users calculate an array in block units, such as in the cache blocking technique for a node with data dependency, the user-specified task dependency and generated data dependency for halo exchange may not identically match. By specifying the `chunksize` clause, the halo region is distributed logically to equal-sized contiguous chunks, and data dependencies for the halo exchange are generated automatically by the XMP runtime system based on the specified chunk size.

Figure 6 presents an example of the `tasklet gmove` directive. In this example, array *A[]* with length three is distributed to three nodes in equal-sized contiguous blocks. This code creates three kinds of tasks. TaskA and taskC are executed on nodes specified by the `on` clause. TaskB is executed on nodes 1 and 2, because these nodes have the specified distributed array *A[0]* or *A[1]* in the associated assignment statement under the `tasklet gmove` directive. There is a flow dependency between taskA and taskB on node 1 by *A[0]*. After the execution of taskA, taskB sends *A[0]* to node 2, which is determined by the distributed array *A[1]*. In node 2, taskB receives *A[0]* from node 1 in *A[1]*. When the receive operation in taskB is finished, taskC is immediately started, because the flow dependency of *A[1]* is satisfied. TaskC sends the *A[1]* to variable *B* of node 3. Because the variable *B* is a local variable for each node, the communication destination is determined from the execution nodes specified by the `on` clause.

3.3 *Prototype Design of Code Transformation*

We have designed a simple code transformation from the code using the proposed directives to the code with XMP runtime calls using MPI and OpenMP. As for a

preliminary evaluation, we have made a hand-translated MPI and OpenMP code by using the proposed transformation.

The `tasklets` directive is converted into the OpenMP `parallel` and `single` directives. The execution node is determined by the `on` clause, which is translated to an `if` statement. The `tasklet gmove` and `tasklet reflect` directives are converted into MPI_Send/Recv(), and these MPI functions are executed in OpenMP tasks with data dependency specified by users. In the case that an MPI blocking call, such as MPI_Send/Recv(), occurs in these codes, a deadlock may occur depending on the task scheduling mechanism, from the combination of MPI and OpenMP. To prevent this deadlock, in the actual implementation we used MPI asynchronous communications, such as MPI_Isend/Irecv(), MPI_Test(), and the OpenMP `taskyield` directive, which makes the current task become suspended at the time point at which it is invoked, and may result in switching to different tasks.

3.4 Preliminary Performance

We measured the performance on the Oakforest-PACS [11] systems at at the Joint Center for Advanced High-Performance Computing (JCAHPC) [9], under cooperation with the Center for Computational Sciences, University of Tsukuba and the Information Technology Center, the University of Tokyo. This system has 8,208 computing nodes, each of which consists of an Intel Xeon Phi (KNL) processor and the Intel Omni-Path architecture as an interconnection. In this evaluation, we selected the Flat and Quadrant modes for KNL. While the Intel Xeon Phi 7250 has 68 cores, a 64 core usage per node is recommended in this system. Some cores are used to assist the OS, interrupt handling, and for communication progress. Moreover, in order to avoid OS jitters, only core 0 is set to receive OS interruptions.

We used blocked Cholesky factorization as our benchmark. It calculates the decomposition of a Hermitian positive-definite blocked matrix into the product of a lower triangular matrix and its conjugate transpose. The calculation consists of four BLAS or LAPACK functions, POTRF, TRSM, GEMM, and SYRK, which are performed in block units. Figure 7 shows the Blocked Cholesky factorization code in the XMP tasklet directive.

We compare the performance in two parallelization approaches, “Parallel Loop” and “Task,” in MPI and OpenMP. The “Parallel Loop” version is the conventional barrier-based implementation, described by work sharing for loops using the `parallel for` directive and independent tasks using the `task` directive without the `depend` clause. Although this version of blocked Cholesky factorization is applied on the overlap of the communication and computation at the process level, it performs the global synchronization in work sharing. The “Parallel Loop” version of the Laplace equation solver does not include the overlap of the communication and computation. The “Task” version is implemented using our proposed model, based on task dependency using the `depend` clause, instead of global synchronization.

```
1  double A[nt][nt][ts*ts], B[ts*ts], C[nt][ts*ts];
2  #pragma xmp nodes P(*)
3  #pragma xmp template T(0:nt−1)
4  #pragma xmp distribute T(cyclic) onto P
5  #pragma xmp align A[*][i][*] with T(i)
6
7  #pragma xmp tasklets
8  for (int k = 0; k < nt; k++) {
9  #pragma xmp tasklet out(A[k][k]) on T(k)
10     potrf(A[k][k]);
11
12 #pragma xmp tasklet gmove in(A[k][k]) out(B) on T(k:)
13     B[:] = A[k][k][:];
14
15     for (int i = k + 1; i < nt; i++) {
16 #pragma xmp tasklet in(B) out(A[k][i]) on T(i)
17         trsm(B, A[k][i]);
18
19 #pragma xmp tasklet gmove in(A[k][i]) out(C[i]) on T(i:)
20         C[i][:] = A[k][i][:];
21     }
22     for (int i = k + 1; i < nt; i++) {
23         for (int j = k + 1; j < i; j++) {
24 #pragma xmp tasklet in(A[k][i], C[j]) out(A[j][i]) on T(j)
25             gemm(A[k][i], C[j], A[j][i]);
26         }
27 #pragma xmp tasklet in(A[k][i]) out(A[i][i]) on T(i)
28         syrk(A[k][i], A[i][i]);
29     }
30 }
```

Fig. 7 Blocked Cholesky factorization code in the XMP `tasklet` directive

We also show the result of these benchmarks implemented by MPI and OmpSs as "Task (OmpSs)." This implementation is described in the `in`, `out`, and `inout` clauses with the OmpSs `task` directive. The `parallel` and `single` regions are not required in the OmpSs programming model. Except for these differences, this is almost the same as the `Task` version.

We evaluated these benchmarks on the following node configurations. For the Oakforest-PACS system, it is on 1–32 nodes, one process per node, 64 cores per process, and one thread per core. The problem size of these benchmarks is set by a matrix size of 32,768 × 32,768 and a block size of 512 × 512 in double precision arithmetic. The matrix is distributed by a two-dimensional block-cyclic data distribution in blocked Cholesky factorization.

Figure 8 illustrates the performance and breakdown of blocked Cholesky factorization on the Oakforest-PACS. The breakdown indicates the average time required

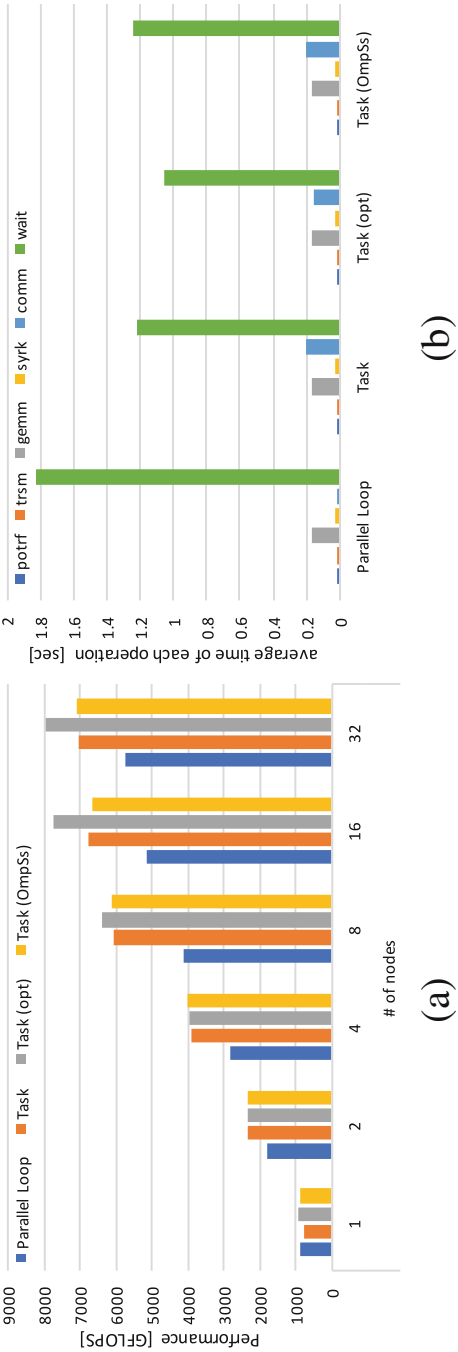

Fig. 8 Performance and breakdown of blocked Cholesky factorization on the Oakforest-PACS system. (**a**) Performance. (**b**) Breakdown at 32 nodes execution

for each operation performed on all threads, because tasks executed on threads differ each time the program is executed. The "wait" in the breakdown represents the waiting time of the thread, including the global synchronization. The "comm" indicates the time from the start of the communication to the end. In Fig. 8a, the "Task" version shows a better performance than the barrier-based "Parallel Loop" implementation. The reason that the "Task" version outperforms the "Parallel Loop" version is that the global synchronization uses a higher cost for the work sharing of loops and among tasks, as shown in Fig. 8b. The relative performance of the "Task" version compared with the "Parallel Loop" version is 123% (Fig. 8).

3.5 Communication Optimization for Manycore Clusters

In the global task parallel programming model, the communication may happen at each pair of tasks between nodes. In order to enable the communication in multithreaded environment, we may use MPI_THREAD_MULTIPLE as the MPI thread-safety level, because tasks executed on threads may communicate simultaneously. We have examined the basic performance of multithreaded communications by using the Ping-Pong benchmark. This benchmark is based on the OSU Micro-Benchmarks 5.3.2 [12] developed by the Ohio State University. we also show the aggregated bandwidth when multiple threads (i.e., two, four, or eight threads) communicate at the same time. Figure 9 illustrates the communication performance on the Oakforest-PACS system. The performance of multithreaded communication with MPI_THREAD_MULTIPLE degrades compared to a single-threaded communication as the number of threads increases. As with the result on the Oakforest-PACS system, the performance of communication on a single thread is better compared to that for multithreaded communication with MPI_THREAD_MULTIPLE. Therefore, the communication performance may be improved if all communications are delegated to the communication thread. To delegate the communications to a single thread, we create a global queue that is accessible by all threads, so that the tasks enqueue the communication requests into this queue and wait for the communication to complete. Meanwhile, the communication thread dequeues the requests for communication to perform the requested communications, and checks the communication completion. The communication thread executes only the communication, and the other threads perform computation tasks.

Figure 8 shows the performance and breakdown of blocked Cholesky factorization with the communication optimization denoted as "Task (opt)." The "Task (opt)" version of blocked Cholesky factorization performs better than the multitasking execution with MPI_THREAD_MULTIPLE. The reason for this is that the communication time decreases compared with the "Task" version, as shown in Figs. 8, because of the use of the communication thread. The relative performances compared with the barrier-based "Parallel Loop" implementation improve to 138% on the Oakforest-PACS systems.

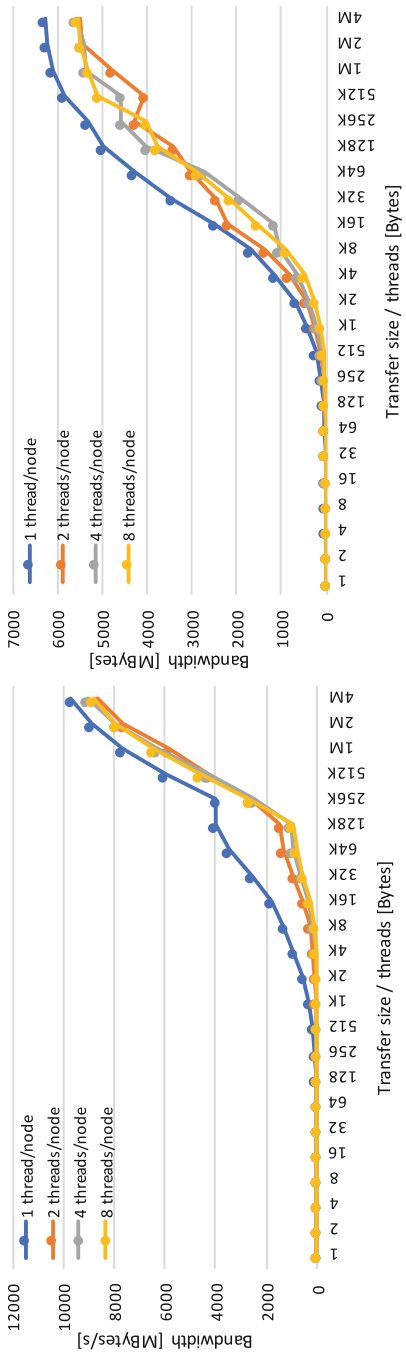

Fig. 9 Performance of the Ping-Pong benchmark on the Oakforest-PACS and COMA systems. (**a**) Oakforest-PACS. (**b**) COMA

4 Retrospectives and Challenges for Future PGAS Models

Since 2007, we have been developing the XcalableMP PGAS language and its reference implementation by the Omni compiler.

In this section, the challenges for future PGAS models are presented with some retrospectives on our project.

4.1 Low-Level Communication Layer for PGAS Model

PGAS is implemented by Remote Memory Access (RMA) providing light-weight one-sided communication and low overhead synchronization semantics. For programmers, both PGAS and RMA are programming interfaces and offer several constructs such as remote read/write and synchronizations.

Remote Direct Memory Access (RDMA) is a mechanism (operation) to access data in remote memory by giving address in (shared) address space. It can be done without involving the CPU or OS at the destination node. Recent advanced interconnect such as Cray Aries interconnect and Fujitsu Tofu of K computer and Tofu-D of Fugaku support remote DMA operations which strongly support efficient one-sided communication.

For the most PGAS runtimes, one-sided communication operations such as Remote Direct Memory Access (RDMA) functions in the MPI are used to implement remote put/get operations in the PGAS languages. Although MPI3 provides several RMA APIs as library interface, the advantages of direct use of RMA/RDMA Operations are as follows:

- Multiple data transfers can be performed with a single synchronization operation.
- Some irregular communication patterns can be more economically expressed.
- The RDMA can be significantly faster than send/receive on systems with hardware support for remote memory access.

We found the multiple data transfers for the stencil computation can be optimized by using a single synchronization operation at the end [13]. As described in Chap. 3, our XMP Coarray were implemented by both MPI and Fujitsu low-level Tofu API. In case of MPI, we used “passive target” mode in MPI one-sided communication. It is noted that the MPI flush operation and synchronization do not sometimes match to implement “sync_images”, and the complex “window” management to expose the memory as a coarray. Finally, Fujitsu RDMA interface is much faster than MPI in the K computer.

Other problem is the communication in the multithreaded environment. As described in the previous sections, we found the performance problem of MPI_THREAD_MULTIPLE. As the connection-less semantics of RDMA would be suited to communications in multithreaded environment, we believe that a new design of low-level communication layer would be a desirable solution in near future.

4.2 XcalableMP as a DSL for Stencil Applications

The Domain Specific Language (DSL) is a promising approach to make the programing easy in a specific domain. Many DSLs such as OpenFOAM in CFD are successful.

Many DSLs are proposed to describe the typical stencil computation. On the other hand, we propose the mixed-language programming with XcalableMP in Chap. 5. Using this model, the main kernel of the computation can be written in XcalableMP and other controls, input/output and house-keeping operation are written by other familiar languages such as Python. In this case, a part of XMP is thought as a kind of DSL to write the stencil computation with global view programming.

The advantages of this approach are as follows:

- By using the XMP global view programming model, the stencil computation can be described in a simple loop based on its original sequential program.
- The stencil communication can be done by the XMP optimized stencil communication runtime [13].
- The advanced optimization of the stencil operations is enabled by a set of the directives for the extended stencil optimization such as a loop unrolling and temporal blocking, added in the latest XcalableMP specification, version 1.4 [14].

4.3 XcalableMP API: Compiler-Free Approach

Although many PGAS languages, such as UPC and Chapel, CAF, have proposed, it is hard to say that they are fully accepted by the community of parallel programming. Recently, the libraries supporting the PGAS model, such as OpenShmem [15], GlobalArray [16], even MPI3 RMA, are getting popular for programming some specific applications. Furthermore, many C++ template-based design for PGAS, such as UPC++ [17], DASH [18], are proposed as a compiler-free approach, as C++ template provides powerful abstraction mechanism. This approach may increase portability, clean separation from base compiler optimization, but a problem is that it is sometimes hard to debug in C++ template once a programmer writes wrong programs.

The approach of extending the language given by the support of the compiler, the compiler-approach, may give:

- A new language, or language extension provides easy-to-use and intuitive feature resulting in better productivity.
- This approach enables the compiler analysis for further optimization, such as removal of redundant sync and selection of efficient communication.

In reality, the compiler-approach is not easy to be accepted for deployment and supports in many sites, resulting in the failure of wide dissemination.

We will have a plan to design the library interface for XcalableMP programming model, XMP API, which is aiming to provide the most equivalent programming functions by the set of libraries.

4.4 Global Task Parallel Programming Model for Accelerators

The task-based programming recently supported in OpenMP 4.0 enables to expose a lot of parallelism by executing several tasks of the program in the form of task-graph. To accelerate the task-based parallel program by accelerators such as GPU and FPGA, it is useful for some tasks frequently executed in parallel to be offloaded to accelerators as an asynchronous task executed by accelerators.

In previous section, the global task parallel programming model is presented. The next step will be that this global task parallel programming model is extended to tasks offloaded to accelerators attached to each node in accelerated clusters.

Exploration of new high-performance architectures from programing model's point of view is an important challenge. Future parallel architecture will be more heterogenous having many kinds of accelerators and devices attached to the nodes and directly connected between accelerators by some dedicated interconnect. To program such a complex and heterogenous parallel system, the global task parallel programming model will give a flexible and decomposable model to exploit such heterogenous high-performance architecture.

References

1. *Flagship 2020 Project (Supercomputer Fugaku)*, https://www.r-ccs.riken.jp/en/overview/exascalepj/
2. H. Murai, M. Sato, An efficient implementation of stencil communication for the XcalableMP PGAS parallel programming language, in *7th International Conference on PGAS Programming Models*, Edinburgh (2013)
3. FUJITSU Ltd., *Development Studio uTofu User's Guide* (2020)
4. RIKEN Advanced Institute for Computational Science (RIKEN AICS), *Fiber Miniapp Suite* (2104), fiber-miniapp.github.io
5. H. Murai, M. Nakao, H. Iwashita, M. Sato, Preliminary performance evaluation of coarray-based implementation of fiber Miniapp suite using XcalableMP PGAS language, in *Second Annual PGAS Applications Workshop (PAW)*, Denver, CO (2017)
6. *CCS QCD Solver benchmark program*, https://www.ccs.tsukuba.ac.jp/qcd/ccsqcdsolverbenchmic/
7. *NTChem Overview*, https://www.r-ccs.riken.jp/software_center/software/ntchem/overview/
8. K. Tsugane, J. Lee, H. Murai, M. Sato Multi-tasking execution in PGAS language XcalableMP and communication optimization on many-core clusters, in *HPC Asia 2018*, Tokyo (2018), pp. 75–85
9. Joint Center for Advanced High Performance Computing (JCAHPC), *Basic Specification of Oakforest-PACS*, http://jcahpc.jp/files/OFP-basic.pdf

10. D. Alejandro, A. Eduard, B. Rosa M, L. Jesus, M. Luis, M. Xavier, P. Judit, OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**, 173–193 (2011)
11. Joint Center for Advanced High Performance Computing (JCAHPC), *Basic Specification of Oakforest-PACS*, http://jcahpc.jp/files/OFP-basic.pdf
12. *OSU Micro-Benchmarks*, http://mvapich.cse.ohio-state.edu/benchmarks/
13. H. Iwashita, M. Nakao, H. Murai, M. Sato, A source-to-source translation of coarray Fortran with MPI for high performance, in *HPC Asia 2018*, Tokyo (2018)
14. *XcalableMP Language Specification v 1.4*, https://xcalablemp.org/download/spec/xmp-spec-1.4.pdf
15. *OpenShmem*, http://www.openshmem.org/site/
16. *Global Arrays*, https://hpc.pnl.gov/globalarrays/
17. Y. Zheng, A. Kamil, M.B. Driscoll, H. Shan, K. Yelick, UPC++: a PGAS extension for C++, in *2014 IEEE 28th International Parallel and Distributed Processing Symposium* (2014), pp. 1105–1114
18. K. Fuerlinger, T. Fuchs, R. Kowalewski, DASH: a C++ PGAS library for distributed data structures and parallel algorithms, in *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Sydney, NSW (2016), pp. 983–990. https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0140