An Efficient Implementation of Stencil Communication for the XcalableMP PGAS Parallel Programming Language

Hitoshi Murai[1] and Mitsuhisa Sato[2]

[1] RIKEN AICS, Kobe, Japan
h-murai@riken.jp
[2] Center for Computational Science, University of Tsukuba, Tsukuba, Japan
msato@cs.tsukuba.ac.jp

Abstract

Partitioned Global Address Space (PGAS) programming languages have emerged as a means by which to program parallel computers, which are becoming larger and more complicated. For such languages, regular stencil codes are still one of the most important goals. We implemented three methods of stencil communication in a compiler for a PGAS language XcalableMP, which are 1) based on derived-datatype messaging; 2) based on packing/unpacking, which is especially effective in multicore environments; and 3) experimental and based on one-sided communication on the K computer, where the RDMA function suitable for one-sided communication is available. We evaluated their performances on the K computer. As a result, we found that the first and second methods are effective under different conditions, and selecting either of these methods at runtime would be practical. We also found that the third method is promising but that the method of synchronization is a remaining problem for higher performance.

1 Introduction

As computer systems become larger and more complicated, for example, with respect to memory hierarchy and interconnect topology, to achieve higher performance, a programming method that can provide users with high productivity and high performance is strongly demanded. Partitioned Global Address Space (PGAS) programming languages, such as XcalableMP (XMP) [22], the coarray feature of Fortran 2008 [18], Unified Parallel C (UPC) [21], Chapel [8], and X10 [5], are considered to meet this demand and have been investigated extensively.

A number of PGAS languages set the goal of supporting a broader range of applications, such as irregular applications having task parallelism that High Performance Fortran (HPF) [14], which is an ancestor of PGAS languages, could never support successfully. However, the situation whereby regular stencil codes, such as reported in [19, 10, 7], are among the most significant goals remains unchanged. Therefore such languages should provide a means for effectively handling stencil communication.

We implemented three types of stencil communication in the Omni XcalableMP compiler that we are currently developing, and the details of these types of stencil communication are presented herein. The first type is based on the derived datatype of Message Passing Interface (MPI) [15]. The second type is based on packing/unpacking buffers, which may be executed in parallel if possible. Finally, the third type is experimental and is based on the extended RDMA interface [9] dedicated for the K computer [16]. The goal of the present study is to explore an optimal method of implementing stencil communications in compilers for PGAS languages.

The contributions of the present paper include:

- Three implementations of stencil communication, including an RDMA-based implementation, for PGAS language compilers, are described.
- Their advantages and disadvantages are discussed based on their evaluation on the K computer.

The remainder of the present paper is organized as follows. Sections 2 and 3 provide a brief overview of the XMP language specification and the Omni XMP compiler, respectively. Sections 4 and 5 describe the proposed implementations of stencil communication, which are evaluated in Section 6. After discussing related research in Section 7, Section 8 presents the conclusion and areas for the future research.

2 XcalableMP

XcalableMP (XMP) is a directive-based language extension for Fortran and C, proposed by the XcalableMP Specification Working Group. XMP supports typical parallelization methods based on the data/task parallel paradigm under the "global-view" model, and enables parallelization of the original sequential code with minimal modification. XMP also includes the coarray feature imported from Fortran 2008 for "local-view" programming. In addition, the combination of OpenMP directives and XMP is to be included in the next update of its specification. In this section, we present a brief overview of the specification of XMP.

The readers can find an example of an XMP program in Figure 8.

2.1 Execution and Memory Model

Execution Model The execution entities in an XMP program are referred to as *XMP nodes* or, more simply, *nodes*. An XMP node is mapped at runtime to a physical computation node on which an MPI process can run with multithreading in hybrid parallelization or with multiple MPI processes in flat parallelization.

The basic execution model of XMP is Single Program Multiple Data (SPMD). Each XMP node starts execution from the same main routine and continues to execute the same code independently (i.e., asynchronously) until an XMP directive, which is *global* and to be executed collectively by all of the nodes, is encountered.

Memory Model Each node has its own memory and can directly access only data contained therein. If a node should access data on a remote node, users must explicitly specify an internode communication with an XMP directive, such as `reflect` described in the following section, in global-view programming or coarrays in local-view.

2.2 Data and Work Mapping

Data Mapping First, an array is *aligned* with a *template*, which is a virtual array, by the `align` directive. Next, the template is *distributed* onto a *node set* in a certain format, such as the block format, the cyclic format, or the block-cyclic format, by the `distribute` directive. As a result, each element of the array is assigned through the distributed template to one or more nodes (Figure 1). The set of local elements of an array logically form a rectangle and is allocated in the local memory.

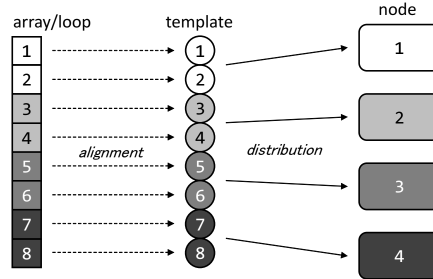


Figure 1: Data and Work Mapping in XMP

Work Mapping An iteration space of a loop nest is, in analogy with an array, "aligned" with a template by the `loop` directive. An aligned loop nest is executed in parallel by the executing nodes.

2.3 Directives for Stencil Communication

2.3.1 The `shadow` Directive

An array distributed in the block or non-uniform block ("gblock") format may have an additional area referred to as *shadow*, which is used as a buffer to communicate with the neighbor elements of each block of the array.

Figure 2 (a) shows the syntax of the `shadow` directive of XMP, which is used to specify the width of the shadow area of each axis of an array[1]. Users can also specify different widths for the lower and upper shadows of an axis.

2.3.2 The `reflect` Directive

Figure 2 (b) shows the syntax of the `reflect` directive of XMP, which is used to update the shadow area of an array with the value of its corresponding reflection source.

Specifying the `width` clause, only a part of the shadow area can be updated. In addition, when the `/periodic/` modifier is specified in the `width` clause, the update is "periodic" along the axis, which means that the shadow object at the global lower (upper) bound is updated with the value of the data object at the global upper (lower) bound.

A communication induced by the `reflect` directive can be *asynchronous* when the `async` clause is specified with the directive. Such asynchronous communications are issued but not completed, along with nonblocking communications of the MPI standard, at the point of the directive to overlap with the following computation.

Figure 3 illustrates how the `shadow` and `reflect` directives work for a one-dimensional array.

2.3.3 The `wait_async` Directive

The `wait_async` directive (Figure 2 (c)) blocks and therefore statements following it are not executed, until all of the asynchronous communications specified by *async-id*s are complete.

[1]When *shadow-width* is of the form "`*`", the entire area of the array is allocated on each node, and all of the area not owned by it is regarded as shadow. This feature is referred to as "full shadow" but is not dealt with in the present paper.

```
[F]  !$xmp shadow array-name ( shadow-width [, shadow-width]... )
[C]  #pragma xmp shadow array-name [shadow-width][[shadow-width]]...
```

where *shadow-width* must be one of:

int-expr
int-expr : *int-expr*
*

(a) the `shadow` directive

```
[F]  !$xmp reflect ( array-name [, array-name]... ) █
                   █ [width ( reflect-width [, reflect-width]... )] [async ( async-id )]
[C]  #pragma xmp reflect ( array-name [, array-name]... ) █
                   █ [width ( reflect-width [, reflect-width]... )] [async ( async-id )]
```

where *reflect-width* must be one of:

[/periodic/] *int-expr*
[/periodic/] *int-expr* : *int-expr*

(b) the `reflect` directive

```
[F]  !$xmp wait_async ( async-id [, async-id ]... ) [on nodes-ref | template-ref]
[C]  #pragma xmp wait_async ( async-id [, async-id ]... ) [on nodes-ref | template-ref]
```

(c) the `wait_async` directive

Figure 2: Syntax of the XMP directives for stencil communication (`[F]` is the line for XMP/-Fortran, and `[C]` the line for XMP/C. █ indicates that the syntax rule continues.)

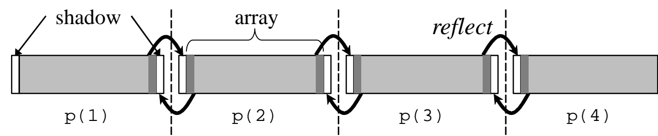


Figure 3: Workings of the `shadow` and `reflect` directives

Note that communications other than those induced by `reflect` can be asynchronous in XMP, and `wait_async` may have to handle them.

3 Omni XcalableMP

Omni XcalableMP is a reference implementation of an XMP compiler that is being developed as an open-source project by the HPCS Laboratory of the University of Tsukuba and Programming Environment Research Team of RIKEN AICS [1].

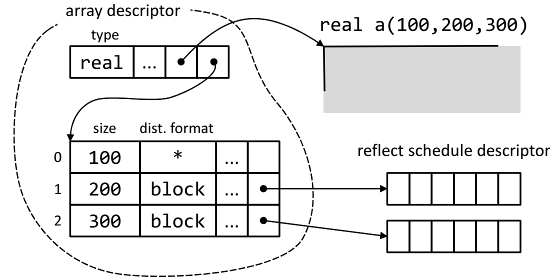


Figure 4: Descriptors in Omni XMP

Omni XMP consists of two major parts: a translator and a runtime library. The translator translates an XMP source program into a program that is in the base language and involves calls to the runtime routines. In particular, each executable directive, such as `reflect` and `wait_async`, in the source program is replaced with a sequence of runtime routine calls. The runtime library is in charge of, for example, parallel execution control, communication and synchronization, and memory management at runtime.

In the current implementation, the runtime library is based on MPI for portability, although those based on other communication libraries such as the extended RDMA interface of the K computer, which is dealt with in Section 5, and GASNet [3] are also being developed or planned.

The current implementation supports platforms of Linux clusters, Cray machines, the K computer, and any other machines on which MPI works.

4 Implementation

We implemented the `reflect` communication using two methods for general (MPI-supported) platforms: One method is based on MPI's derived datatype, and the other method is based on packing/unpacking buffers.

The XMP runtime system autonomously determines at runtime which of the two methods is used for stencil communications. In addition, users can explicitly specify the method with an environment variable.

4.1 Reflect Schedule Descriptor

The Omni XMP runtime system manages a descriptor of each distributed array to be referenced as necessary by the runtime library. The lifetime of the descriptor is the same as that of the corresponding array. In addition to this array descriptor, if a shadow area is declared for a dimension of an array, the runtime system creates a *reflect schedule descriptor (RSD)*, which stores information on the schedule of a `reflect` communication for the dimension, and links the RSD from the array descriptor (Figure 4). Once created, an RSD is reused repeatedly unless the schedule is changed by another `reflect` directive with different clauses specified. Table 1 shows the components of the RSD.

Table 1: Components of the reflect schedule descriptor

Type	Name	Description
`int`	`lo_width` `hi_width`	latest widths
`int`	`is_periodic`	latest periodic flag
`MPI_Datatype`	`datatype_lo` `datatype_hi`	MPI vector datatype
`MPI_Request`	`req[4]`	MPI request handles for upper/lower send/recv
`void*`	`lo_send_buf` `lo_recv_buf`	buffers for lower shadow
`void*`	`hi_send_buf` `hi_recv_buf`	buffers for upper shadow
`void*`	`lo_send_array` `lo_recv_array`	target positions in array for upper shadow
`void*`	`hi_send_array` `hi_recv_array`	target positions in array for upper shadow

`int`	`count`
	`blocklength`
	`stride`
		components of vector (used in pack/unpack)

`int`	`lo_rank` `hi_rank`	MPI ranks of neighboring nodes

4.2 Method 1: Derived Datatype

Any `reflect` communication can be performed as a point-to-point nonblocking communication of a message of type *vector* and length one, where vector is one of MPI's built-in derived datatypes consisting of equally spaced blocks and constructed by the function `MPI_TYPE_VECTOR`.

The vector datatype has three components: *count* for the number of blocks; *blocklength* for the number of elements in each block; and *stride* for the number of elements between start of each block.

The count, blocklength, and stride of the vector for `reflect` in the k'th dimension of an N-dimensional array are calculated as follows[2]:

$$
\begin{aligned}
count &= lsize_{k+1} \times \cdots \times lsize_{N-1} \\
blocklength &= lsize_0 \times \cdots \times lsize_{k-1} \times shadow_k \\
stride &= lsize_0 \times \cdots \times lsize_k
\end{aligned}
$$

where $lsize_i$ and $shadow_i$ represent the local size, which is the size of elements resident on each node, and the width of the lower or upper shadow area, in the i'th dimension of the array, respectively. Note that the local size includes the size of the shadow area.

The schedule of the nonblocking communication of the vector is bound to a *persistent communication request*, which is stored in the RSD and is used to initiate and complete persistent communication in functions `MPI_Startall` and `MPI_Waitall`, respectively (Figure 5).

Note that a schedule is created for each dimension of the array but, in the current implementation, persistent communications for all dimensions are issued asynchronously in a batch. This means that shadow areas at the corner boundaries of an array may not be updated properly, and, therefore, the nine-point difference cannot be handled. This problem can be resolved

[2]This applies to the Fortran-style column-major ordering of array elements. These calculations for C can be obtained easily but are not presented herein

```
1  // create datatypes
2  for (i = 0; i < ndims; i++){
3   MPI_Type_vector(count, blocklength*lwidth, stride, MPI_BYTE, &reflect->dt_lo);
4   MPI_Type_commit(&reflect->dt_lo);
5   ...
6  }
7
8  // initiate persistent comms.
9  for (i = 0; i < ndims; i++){
10  MPI_Recv_init(rbuf_lo, 1, reflect->dt_lo, src, tag, comm, &reflect->req[0]);
11  ...
12  MPI_Send_init(sbuf_hi, 1, reflect->dt_hi, dst, tag, comm, &reflect->req[3]);
13 }
14
15 // do persistent comms.
16 MPI_Startall(4*ndims, reflect->req);
17 MPI_Waitall(4*ndims, reflect->req, status);
```

Figure 5: Overview of derived-datatype method

easily by issuing persistent communications for each dimension synchronously and in sequence. However, for asynchronous `reflect` (described in Section 4.4), communications between ordinal neighbor nodes should be implemented in order to properly update the shadow area. The pack/unpack and the RDMA methods described in the following sections also have the same problem.

4.3 Method 2: Pack/Unpack

The method of communication of a message of type vector performed by the MPI library is implementation-dependent. One implementation can pack a vector into a contiguous buffer before sending data, whereas another implementation might send blocks of a vector one by one without packing. In general, internal packing/unpacking in sending/receiving a vector should be considered to be neither fully optimized nor multithreaded even in a multicore environment. Note that it is theoretically possible to parallelize packing/unpacking vectors, whereas this is not possible for a general datatype.

In order to achieve higher performance primarily in multicore environments, routines for packing/unpacking vectors are multithreaded using an OpenMP directive. Note that the specification states that an XMP directive is single-threaded and therefore an implementation can use multithreading to parallelize the corresponding runtime library routines.

However, such parallelization is effective only when more than one processor core is available in an XMP node (i.e., when using hybrid parallelization). Therefore the Omni XMP runtime system determines whether the packing/unpacking operation is to be executed in parallel, using an OpenMP API runtime library routine `omp_get_num_procs`, which returns the number of processors (cores) available to the program.

Figure 7 shows the internal packing routine `_XMPF_pack_vector` in Omni XMP, where variables `count`, `blocklength`, and `stride` are the same as those of the derived-datatype method. The loop is executed in parallel only if the number of available processor cores is greater than one and the amount of packing/unpacking operation is large enough for parallelization. The `THRESHOLD` variable indicates the threshold of the amount for parallelization, and the appropriate value of `THRESHOLD` depends on the environment.

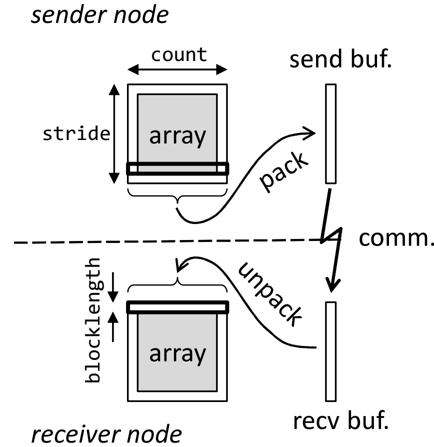


Figure 6: Packing/Unpacking a vector in `reflect`

```
void _XMPF_pack_vector(char * restrict dst, char * restrict src,
                       int count, int blocklength, int stride){

  if (_xmp_omp_num_procs > 1 && count * blocklength > THRESHOLD){
#pragma omp parallel for
    for (int i = 0; i < count; i++){
      memcpy(dst + i * blocklength, src + i * stride, blocklength);
    }
  }
  else {
    for (int i = 0; i < count; i++){
      memcpy(dst + i * blocklength, src + i * stride, blocklength);
    }
  }

}
```

Figure 7: Packing routine

The communication buffer used for packing/unpacking in this method is managed by the runtime system. In the current implementation, once allocated for the dimension of an array, the communication buffer persists and is reused repeatedly for the lifetime of the array.

4.4 Asynchronous Communication

As shown in Section 2.3.2, a `reflect` communication can be asynchronous when the `async` clause specified in a `reflect` directive.

Such an "asynchronous `reflect`" is handled by the Omni XMP runtime system through MPI request handles associated with the nonblocking communications issued for it. The asynchronous `reflect` proceeds at runtime as follows:

1. At a `reflect` directive, a set of nonblocking communications is issued, and their request handles are stored in the *asynchronous communication table (ACT)*, which is a hash table

with *async-id*s as the hash keys.
2. The communications proceed while possibly overlapping some computations.
3. At an `wait_async` directive, the ACT is retrieved with the specified *async-id* to obtain the corresponding request handle, and issues `MPI_Waitall` to complete the nonblocking communications associated with the request.

Note that the `wait_async` directive is used to complete asynchronous communications other than `reflect`, and, therefore, the above mechanism is designed to be applicable to any asynchronous communications in XMP.

In the current implementation, asynchronous `reflect` is performed in the derived-datatype method described in Section 4.2, because issuing a nonblocking communication as early as possible without packing/unpacking in order to facilitate overlapping the following computation is advantageous for achieving high performance.

5 RDMA-based Experimental Implementation

In this section, we present an experimental implementation of the `reflect` directive based on the extend RDMA interface of the K computer[3].

5.1 The Extended RDMA Interface

The MPI library of the K computer and FUJITSU's PRIMEHPC FX10 supercomputer provides users with the *extended RDMA interface*. The interface consists of a number of functions[4] that enable inter-node communication that makes the most of the underlying interconnect hardware, such as Network Interface Controllers (NICs).

When implementing `reflect` communications using RDMA writes of this interface, the following items must be considered.

- An array must be registered to the system and associated with a memory ID using the `FJMPI_Rdma_reg_mem` function, in advance of being accessed through this interface. In the current implementation, all of the distributed arrays with shadow are registered to be (possibly) accessed through the interface.
- The array must be distributed onto the entire node set that corresponds to MPI_COMM_WORLD, because the target process of RDMA is identified with the rank in MPI_COMM_WORLD.
- The availability for the RDMA writes, i.e., whether the shadow areas on the neighboring nodes are ready to be updated, must be explicitly confirmed by each node before issuing the RDMA writes, which means that synchronizations are needed before `reflect`.
- Completion of the RDMA writes, i.e., whether the shadow areas on the neighboring nodes have been updated, must be explicitly confirmed by each node using the `FJMPI_Rdma_poll_cq` function, which means that synchronizations are needed after `reflect`.
- A *tag* that is an integer from 0 to 14 can be assigned to an RDMA in order to identify the RDMA. Since an async-id is used as the tag in the asynchronous mode, the value of the async-id is restricted to the 0 to 14 range.

The third and fourth items are due to the collectiveness of the `reflect` communication.

[3]The implementation is experimental because this implementation has some limitations (e.g., the number of arrays having shadow areas) that are derived from those of the extend RDMA interface and has not yet been released.

[4]These functions are based on a low-level communication library dedicated to the K computer and FX10.

5.2 Method 3: RDMA

Normal Mode A normal `reflect` communication based on the extended RDMA interface is performed in the following steps.

1. Each node waits until all of the nodes reach this point (barrier synchronization);
2. issues an RDMA write for each block of the vector;
3. polls its NICs until all of the RDMA writes issued by the node are completed; and
4. waits until all of the nodes reach this point (barrier synchronization).

The first barrier synchronization guarantees that the neighboring nodes are available, and the second barrier synchronization guarantees that all of the actions involving the communication on both the local and remote nodes are completed.

The reason for the lack of packing/unpacking is that the latency of RDMA writes is sufficiently low and the overhead of issuing multiple RDMA writes is smaller than that of packing/unpacking buffers.

Asynchronous Mode Steps 1 and 2 above are performed by `reflect`, and steps 3 and 4 above by `wait_async`, with the following differences. At `reflect`, RDMA writes are issued while setting the async-id as a tag, and the number of RDMAs issued for the async-id is stored in ACT. At `wait_async`, the NIC is polled until as many RDMAs as extracted from ACT are completed.

6 Evaluation

Using XMP, we parallelized a prototype of the dynamical core of a climate model for large eddy simulation, SCALE-LES [19], which is a typical five-point stencil code in Fortran (Figure 8), and ran the prototype on the K computer [16] in order to evaluate the performance of each implementation of `reflect`. The performance of an MPI-based implementation was also evaluated for comparison. The language environment used was K-1.2.0-13. The problem dimensions were 512×512 horizontally and 128 vertically, and the execution time was measured for 500 time steps.

In this evaluation, we assigned one XMP node to one compute node of the K computer, where intra-node thread-level parallelism can be automatically extracted from node programs by the compiler. The condition for parallelizing packing/unpacking buffers in the pack/unpack method was that the count of a vector (`count` in Figure 7) was more than eight times greater than the number of available cores (`_xmp_omp_num_procs` in Figure 7), i.e., more than eight blocks per thread. Therefore, the length of each block (`blocklength` in Figure 7) was not considered in this evaluation.

For clarify, the evaluation results are presented in three graphs in Figure 9. For the purpose of comparison, some results are presented in more than one graph. The vertical axes in these graphs indicate the speedup of the execution time, relative to that on a single node, and the horizontal axes in these graphs indicate the number of nodes. The computation times of these implementations are approximately equal because their computation codes generated by Omni XMP are identical and are nearly equivalent to that of MPI . Therefore, the difference in the execution time comes from the difference in the communication time (Table 2).

Figure 9 (a) shows the performance of normal-mode `reflect` communications, where MPI indicates the results obtained for the hand-coded MPI version, XMP-dt indicates the results obtained for the derived-datatype method, and XMP-pack indicates the results obtained for

```
1  !$xmp nodes p(N1,N2)
2  !$xmp template t(IA,JA)
3  !$xmp distribute t(block,block) onto p
4     ...
5     real(8) :: dens(0:KA,IA,JA)
6     ...
7  !$xmp align (*,i,j) &
8  !$xmp&   with t(i,j) :: dens, ...
9  !$xmp shadow (0,2,2) :: dens, ...
10    ...
11 !$xmp reflect (dens, ...) width &
12 !$xmp&     (0,/periodic/2,/periodic/2)
13    ...
14 !$xmp loop (ix,jy) on t(ix,jy)
15          do jy = JS, JE
16           do ix = IS, IE
17            ...
18            do kz = KS+2, KE-2
19             ... dens(kz,ix+1,jy) + ...
20             ...
21            end do
22            ...
23           end do
24          end do
```

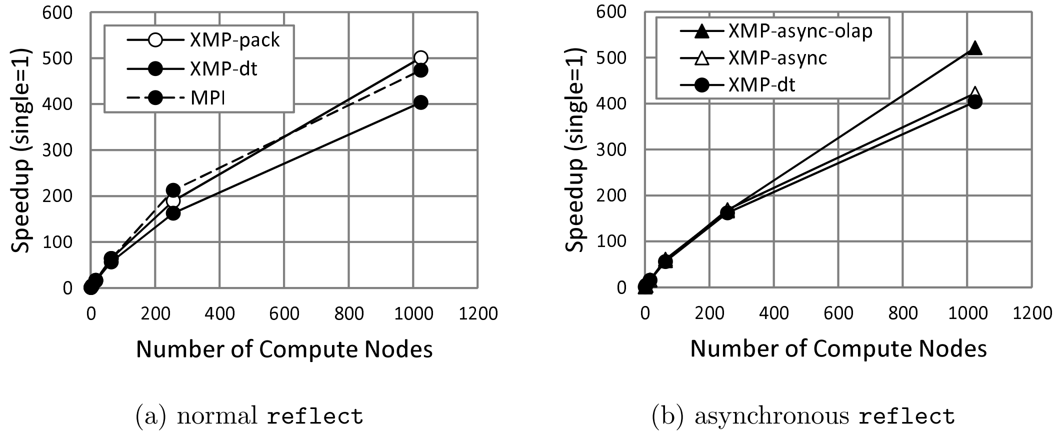
Figure 8: Code snippet of the target climate model

Table 2: Breakdown of the execution time (in seconds)

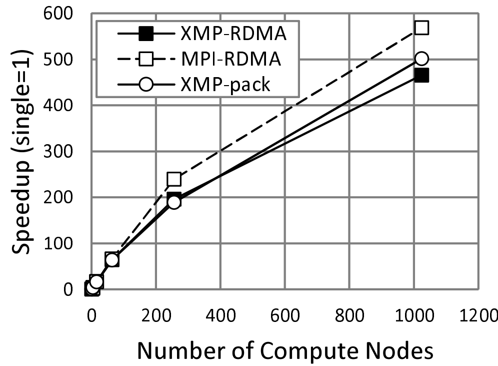
#nodes	4		16		64		256		1024	
	comm.	comp.	comm.	comp.	comm.	comp.	comm.	comp.	comm.	comp.
XMP-pack	8.98	413.1	7.09	102.3	4.95	23.3	4.28	5.35	2.46	1.17
XMP-dt	16.77	413.7	15.79	102.5	8.94	23.3	5.99	5.22	3.30	1.21
XMP-RDMA	7.19	415.4	7.04	101.0	4.80	23.4	4.06	5.22	2.79	1.12
XMP-async	29.50	416.9	15.47	103.3	8.35	23.2	5.48	5.29	3.05	1.26
MPI	15.39	423.6	8.82	100.0	5.47	23.0	3.61	4.98	4.16	N/A
MPI-RDMA	8.39	421.3	2.58	100.1	1.09	23.0	0.64	5.00	1.99	1.21

the pack/unpack method. The pack/unpack method is comparable in performance to the MPI version oand is faster than the MPI version for the 1,024-node execution. However, the results might depend on the fast inter-core hardware barrier of SPARC64 VIIIfx [24]. In fact, we observed that the pack/unpack method is not as effective for an average Linux cluster, as campared to the K computer. On the other hand, the derived-datatype method is slower than MPI. We verified that the derived-datatype method is faster than both the pack/unpack method and MPI in the flat-parallel environment. The results are not presented herein because of space limitations.

Figure 9 (b) shows the performance of asynchronous-mode `reflect` communications, where XMP-dt indicates the results obtained for the synchronous-mode derived-datatype method (for comparison), XMP-async indicates the results obtained for the asynchronous-mode `reflect` that do not overlap with the computations, and XMP-async-olap indicates the results obtained for as much part of the asynchronous-mode `reflect` as possible overlapped with the computations. The overhead introduced for asynchronous communication, such as management and



(a) normal `reflect` (b) asynchronous `reflect`



(c) RDMA-based `reflect`

Figure 9: Evaluation results on the K computer

retrival of ACT, is not so large and the performance is improved significantly by overlapping communication with computation in the 1,024-node execution.

Figure 9 (c) shows the performance of RDMA-based `reflect` communications, where MPI-RDMA indicates the results obtained for the hand-coded RDMA-based version, XMP-pack indicates the results obtained for the pack/unpack method (not based on RDMA, for comparison), and XMP-RDMA indicates the results obtained for the RDMA-based method. The experimental implementation is slower than both the hand-coded RDMA-based implementation and the pack/unpack implementation because the barrier synchronizations before issuing RDMA writes and after completing RDMA writes are too strong to perform stencil communication efficiently. Actually, in the hand-coded implementation, point-to-point synchronizations between neighboring nodes are used instead of barrier synchronizations. In the future, synchronizations performed in RDMA-based `reflect` should be weakened in order to achieve higher performance.

7 Related Research

The `reflect` directive and its asynchronous mode of XMP originates from HPF/JA, which is an extension of High Performance Fortran for accelerating real-world applications [12, 20]. The function of partial reflection was first supported by HPF/SX V2 [17] and HPF/ES [23], the HPF compiler for NEC's SX-series supercomputers and the Earth Simulator, respectively, and later by the dHPF compiler developed by Rice University [6]. Since there is no specification for periodic stencil communication in either the HPF standard or the HPF/JA specification, to our knowledge, no compilers for HPF or HPF-like languages have supported periodic stencil communication yet. On the other hand, a region-based parallel language ZPL that supports periodic stencil communication has been reported [4].

The optimization of stencil communication in HPF is described in a previous study [13], in which a method of generating communications based on realignment was proposed and compile-time optimizations for multidimensional stencil communications were presented.

In [2, 11], implementations of mesh-based regular applications with coarrays, which is a one-sided communication feature from Co-Array Fortran or Fortran 2008, are compared with implementations of mesh-based regular applications with MPI, from the viewpoints of, for example, memory layout and the usage of communication buffers. Stencil communications based on coarrays were demonstrated to be effective in mesh-based regular applications and could, in some cases, outperform stencil communications based on MPI.

8 Conclusions and Future Research

We implemented three methods for stencil communication in the Omni XMP compiler. The first method based on derived-datatype messaging is simple and general, and could be efficient depending on the implementation of the underlying MPI library. The second method is based on packing/unpacking and has the advantage of being multithreaded in multicore environments. The third method, which is experimental and is based on the extended RDMA interface of the K computer, may be able to achieve higher performance, but at present has approximately the same performance as the second method because of exceedingly strong synchronizations.

Areas for future research include:

- managing `reflect` communications from/to ordinal neighbor nodes properly in nine-point difference stencil codes;
- setting an appropriate threshold for parallelizing packing/unpacking buffers in the pack-/unpack method;
- improving the performance of the RDMA-based method by reducing the strength of synchronizations; and
- providing a more portable and efficient implementation based on the one-sided communication of MPI-3.

Acknowledgements

The results were obtained in part using the K computer at the RIKEN Advanced Institute for Computational Science. The original code of the climate model used in the evaluation and its RDMA-based implementation were provided by Team SCALE of RIKEN AICS.

References

[1] Omni XcalableMP Compiler. `http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/xcalablemp/`.

[2] Richard Barrett. Co-array Fortran Experiences with Finite Differencing Methods. In *The 48th Cray User Group meeting, Lugano, Italy*, 2006.

[3] D. Bonachea. GASNet specification. Technical report, University of California, Berkeley (CSD-02-1207), October 2002.

[4] Bradford L. Chamberlain. *The Design and Implementation of a Region-Based Parallel Language.* PhD thesis, University of Washington, November 2001.

[5] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Christoph von Praun, Vijay Saraswat, and Vivek Sarkar. X10: An Object–oriented Approach to Non–Uniform Cluster Computing. In *Proc. OOPSLA 05*, 2005.

[6] D. Chavarria-Miranda and J. Mellor-Crummey. An Evaluation of Data-Parallel Compiler Support for Line-Sweep Applications. *J. Instruction Level Parallelism*, 5, 2003.

[7] Collins, William D and Bitz, Cecilia M and Blackmon, Maurice L and Bonan, Gordon B and Bretherton, Christopher S and Carton, James A and Chang, Ping and Doney, Scott C and Hack, James J and Henderson, Thomas B and others. The Community Climate System Model Version 3 (CCSM3). *J. Climate*, 19(11):2122–2143, 2006.

[8] Cray Inc. Chapel Language Specification 0.93. `http://chapel.cray.com/spec/spec-0.93.pdf`, 2013.

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

[10] Furumura, Takashi and Chen, Li. Parallel simulation of strong ground motions during recent and historical damaging earthquakes in Tokyo, Japan. *Parallel Computing*, 31(2):149–165, 2005.

[11] Manuel Hasert, Harald Klimach, and Sabine Roller. CAF versus MPI - Applicability of Coarray Fortran to a Flow Solver. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 228–236, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] Japan Association of High Performance Fortran. HPF/JA Language Specification. `http://www.hpfpc.org/jahpf/spec/hpfja-v10-eng.pdf`, 1999.

[13] Tsunehiko Kamachi, Kazuhiro Kusano, Kenji Suehiro, and Yoshiki Seo. Generating Realignment-Based Communication for HPF Programs. In *Proc. IPPS*, pages 364–371, 1996.

[14] Ken Kennedy, Charles Koelbel, and Hans Zima. The Rise and Fall of High Performance Fortran: An Historical Object Lesson. In *Proc. 3rd ACM SIGPLAN History of Programming Languages Conf. (HOPL-III)*, pages 7–1–7–22, San Diego, California, June 2007.

[15] Message Passing Interface Forum. MPI: A Message Passing Interface Standard Version 3.0. `http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, 2012.

[16] Hiroyuki Miyazaki, Yoshihiro Kusano, Naoki Shinjou, Fumiyoshi Shoji, Mitsuo Yokokawa, and Tadashi Watanabe. Overview of the K computer. *FUJITSU Sci. Tech. J.*, 48(3):255–265, 2012.

[17] Hitoshi Murai, Takuya Araki, Yasuharu Hayashi, Kenji Suehiro, and Yoshiki Seo. Implementation and Evaluation of HPF/SX V2. *Concurrency and Computation — Practice & Experience*, 14(8–9):603–629, 2002.

[18] Robert W. Numrich and John Reid. Co-arrays in the next Fortran Standard. *ACM Fortran Forum*, 24(2):4–17, 2005.

[19] Team SCALE. SCALE-LES. `http://scale.aics.riken.jp/scale-les/`.

[20] Yoshiki Seo, Hidetoshi Iwashita, Hiroshi Ohta, and Hitoshi Sakagami. HPF/JA: extensions of High Performance Fortran for accelerating real-world applications. *Concurrency and Computation — Practice & Experience*, 14(8–9):555–573, 2002.

[21] UPC Consortium. UPC Specifications, v1.2. Technical report, Lawrence Berkeley National Lab

(LBNL-59208), 2005.

[22] XcalableMP Specification Working Group. XcalableMP Specification Version 1.1. `http://www.xcalablemp.org/xmp-spec-1.1.pdf`, 2012.

[23] Takashi Yanagawa and Kenji Suehiro. Software System of the Earth Simulator. *Parallel Computing*, 30(12):1315–1327, 2004.

[24] Toshio Yoshida, Mikio Hondo, and Ryuji Kan Go Sugizaki. SPARC64 VIIIfx: CPU for the K computer. *FUJITSU Sci. Tech. J.*, 48(3):274–279, 2012.