

Programmability and Performance of New Global-View Programming API for Multi-Node and Multi-Core Processing

Yugo Sakaguchi
Graduate School of Science and Technology
Seikei University
Tokyo, Japan
dm186204@cc.seikei.ac.jp

Hiroko Midorikawa
Graduate School of Science and Technology
Seikei University
Tokyo, Japan
midori@st.seikei.ac.jp

Abstract—Various partitioned global address space (PGAS) languages capable of providing global-view programming environments on multi-node computer systems have been proposed to improve programming productivity in high-performance computing. However, several PGAS languages often require a detailed description of the remote data access, similar to descriptions used in message passing interface one-sided communications. Some PGAS languages have limitations pertaining to remote data access and recommend their local-view programming models, rather than the global-view ones, due to performance-related reasons. In this study, we propose SMint, which is an application programming interface that provides a global-view programming model with a software distributed shared memory mSMS as the runtime. Using stencil computation as a typical processing method, the performance and programmability of SMint have been compared with those of XcalableMP and Unified Parallel C, which are well-known examples of PGAS languages based on the C language. It was found that SMint achieved the best performance under the ideal global-view programming model.

Keywords—PGAS, directive-based language, API, global-view programming, global address space, parallel language, software distributed shared memory, cluster, shared memory programming, multi-node processing, programming model

I. INTRODUCTION

In high-performance computing, parallel programming with MPI+X (OpenMP [1], OpenACC [2]) is widely used in order to extract the best performance from multiple computing nodes and multiple cores within such nodes. On the other hand, various languages and application programming interfaces (APIs), generally referred to as partitioned global address space (PGAS) models, have been proposed [3–5] to improve the low productivity of program model that uses a message passing interface (MPI) distributed memory model. The PGAS languages usually provide a global-view programming environment by enabling the use of global data arrays and global indexes over multiple computing nodes, unlike the local-view programming model used in an MPI. However, many PGAS languages have limitations pertaining to the accessible range of remote data. To access the remote data, some of these languages also require special descriptions, such as coarray descriptions [6], which involve specifying the ID number of the node containing the data to be accessed. This is significantly different from the

typical shared-memory parallel programming model but is rather similar to the descriptions of the one-sided communication (get/put) seen in MPIs. In most of the PGAS languages, a global address space is not provided as part of the execution processes in a true sense, but, the special PGAS APIs mentioned above are converted to lower-level communication library function calls (e.g., an MPI get/put) by a dedicated PGAS compiler. In other words, there are few systems that allow all node processes to share the same address space and enable access through C-language pointers and addresses. Moreover, due to performance- and implementation-related issues, in practice, several PGAS languages often recommend using explicit descriptions of data communications and descriptions using a local-view model similar to those in MPIs.

XcalableMP (XMP) [4], which is an example of a PGAS language, is an extension of the C language (or Fortran), and has a directive-based API. It extends C to offer for-loop parallelization, data distributed mapping, block assignment statements for array data, and related functionalities. Unified Parallel C (UPC) [5,10] is another PGAS language based on C. It includes new syntax and grammar, such as *upc_forall* statements, and has loop parallelization and distributed data mapping functions. It also introduces a global array data declaration statement with distributed mapping specifiers.

We use the mSMS distributed shared memory system [11–13] as a runtime in order to achieve a global-view programming environment across multiple nodes. Three programming APIs are available in mSMS: (1) a C program using SMS library functions, (2) MpC-language descriptions introducing the global data declarative statement “shared” for easily distributable mapping of global shared data in multiple nodes [13], and (3) SMint [14], a directive-based API which enables multi-node-multi-core parallel processing by adding *#pragma* to a sequential C program.

Table 1 shows a comparison of the three aforementioned languages. In XMP, the possible processing that can be described in the global view model is very limited because of the limitations in the accessible global data area. C pointers for global data are not available in XMP, although it is based on C. On the other hand, in SMint and UPC, there are no restrictions on accessible global data area, and programming with pointers is also possible. In SMint, ordinary pointers used in C programming are available to point both global and local data.

Thus, it provides a seamless data access environment without changing data access APIs according to the data location (global or local). The mSMS runtime determines whether the data to be accessed is remote or local and fetches data from the appropriate remote node at runtime. In UPC, on the other hand, pointers pointing to global data and to local data are defined clearly as different data types. The UPC pointer variable itself is also distinguished as either a shared pointer or a local pointer; hence, there are four types of pointers. By using this explicit distinction, the UPC compiler can statically distinguish global and local data accesses. The compiler statically converts global data accesses to data transfer descriptions using appropriate communication mechanisms. Type conversion from local to global is therefore accompanied by an overhead.

Predominantly, both XMP and UPC depend on conversion of their programs to ones with low-level communication descriptions, based on static analysis by a dedicated compiler. Thus, the load on their runtime system is smaller than that on mSMS runtime systems. On the other hand, SMint has almost no syntax extensions from the C language, and it mainly uses a simple translator to insert SMS library functions into a C program. It uses a general C compiler (gcc). The underlying mSMS distributed shared memory system performs dynamic processing at runtime.

TABLE I. COMPARISON OF UPC, XCALABLEMP, AND SMINT

	UPC	XcalableMP	mSMS (+MpC)	SMint
Access to global data	Anywhere (shared data)	sleeve or gmove	Anywhere	Anywhere
Use pointer	Possible	Impossible	Possible	Possible
Runtime load	Light	Light	Heavy	Heavy
Dedicated Compiler	Required	Required	Unnecessary (Simple translator)	Simple translator
Directive-based programming	Not Available	Available	Not Available	Available

In this study, we compare the programmability and performance of SMint, XMP, and UPC as PGAS languages that support global-view models based on the C language. Stencil computation, which is one of the typical scientific computations, is implemented for the three languages with a global-view model, for the evaluation.

II. PARALLEL PROGRAMMING IN SMINT/MSMS

The distributed shared memory system mSMS [12] enables users to choose one of three APIs depending on the application type and their requirements [14]. All the APIs are converted into C programs using the SMS library functions shown at the bottom of Fig. 1.

A. C Programs from the use of SMS Library Functions

The first API is a C program using SMS library functions. Because `sms_rank` and `sms_nprocs`—corresponding to the MPI rank number and number of processes, respectively—are available, different processing actions for each node can be described. Fig. 2 shows a matrix-vector product program. Global data shared across multiple nodes is dynamically allocated using `sms_alloc` or `sms_mapalloc`. While `sms_alloc` allocates global data of a specified size to a specified node, `sms_mapalloc` is

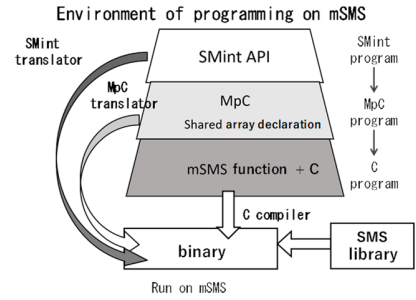


Fig. 1 Programming environment on mSMS

```
#include <sms.h> //C program by using SMS library functions
#define N ...
int main(int argc, char *argv[])
{ int size, st, ed; // Area of each node
  double *vec1, *vec2; // Pointers for 1D array vec1[N] and vec2[N]
  double (*array)[N]; // Pointer for 2D array array[N][N]
  // dim: size of array, div: division number of distribution map
  int dim[3]={N, N, 1}, div[3]={1, 1, 1};
  sms_startup(&argc, &argv);

  vec1 = (double*)sms_alloc(sizeof(double), N, 0); // allocation vec1[N] to node0
  vec2 = (double*)sms_alloc(sizeof(double), N, 1); // allocation vec2[N] to node1
  div[0]=sms_nprocs; // Split array into bands, distributed mapping to all nodes
  array= (double(*)[N]) sms_mapalloc(dim, div, sizeof(double), 0, sms_nprocs);

  size=N/sms_nprocs;
  st=size * sms_rank; ed=size * (sms_rank+1); //Area of each node
  #pragma omp parallel for // Multithreaded parallel execution in each node
  for(i=st; i<ed; i++) { // Parallel execution of for(i=0; i<N; i++) in all nodes
    for(k=0; k<N; k++) vec2[i]= array[i][k] * vec1[k]; // Matrix-vector multiplication
  }
  sms_barrier();
  sms_shutdown();
}
```

Fig. 2 C program with SMS library functions

```
#include <mpc.h> // C program by using MpC (shared data declaration)
#define N ...
shared double vec1[N] ::[1](0,1); // Mapping to node0
shared double vec2[N] ::[1](1,1); // Mapping to node1
shared double array[N][N] ::[NPROCS][0,NPROCS];
//Distributed mapping to all nodes

int main(int argc, char *argv[])
{ int size, st, ed; //Area of each node

  mpc_init(&argc, &argv);

  size=N/NPROCS;
  st=size * MYPID; ed=size * (MYPID+1); //Area of each node
  #pragma omp parallel for //Multithreaded parallel execution in each node
  for(i=st; i<ed; i++){ // Parallel execution of for(i=0; i<N; i++) in all nodes
    for(k=0; k<N; k++)
      vec2[i]= array[i][k] * vec1[k]; //Matrix-vector multiplication
  }
  mpc_barrier();
  mpc_exit();
}
```

Fig. 3 MpC program with shared data declaration

mainly used for distributed mapping of global data arrays. It specifies the number of fractions for each dimension of the array and performs distributed mapping of the array data cyclically over the multiple specified nodes [13].

B. MpC Programs with Global Data Type Array Declaration

The second API is an MpC program that can use multidimensional array declarations with data distribution mapping specifications, as shown in Fig. 3. An array declaration statement concerning shared data in MpC programming [11] is converted to `sms_mapalloc`. It is known that MpC is a minimal extension of C [13] and that it enables global-type array

declarations using *shared*. In scientific simulations, array descriptions are often preferred to dynamic allocation, such as *sms_alloc* or *sms_mapalloc*. The MpC program is converted to an equivalent C program, as shown in Fig. 2, by the MpC translator.

C. SMint Incremental Programming

The third API, SMint [14], is a directive-based API. As with OpenMP or OpenACC, adding pragma SMint to the for-loop statements in sequential programs enables conversion of these programs to multi-node parallel programs. SMint provides incremental programming from sequential programs in a simple way. Moreover, data localizing instructions (*copyin*, *copyout*, *copy*, *create*, etc.) that perform batch data transfer from or to a remote node, before or after multi-node parallel sections can be added. Using this specification, necessary data can be prefetched to the local node prior to the start of the parallel section, which boosts the performance of computation in the parallel section. It also improves the efficiency of data consistency management and local node cache handling at the end of parallel sections. In a case without such data localizing specification, mSMS detects remote data accesses at runtime and fetches the required data from the remote node in SMS-pagesize units. Fig. 4 shows the SMint program. Fig. 4 shows a different version of the matrix-vector product programs shown in Figs. 2 and 3. In Fig. 4, *vec2* is distributed across all nodes and remote data *vec1* is prefetched using the *copyin* instruction.

```

#include <smint.h> // Program by using #pragma SMint copyin
#define N ...
#pragma SMint shared ::[(0,1)]; // Mapping to node0
double vec1[N];
#pragma SMint shared ::[NPROCS](0, NPROCS); // Distributed mapping to all nodes
double vec2[N];
#pragma SMint shared ::[NPROCS]{}(0, NPROCS); // Distributed mapping to all nodes
double array[N][N];

int main(int argc, char *argv[])
{
    // Prefetch vec1(mapped to node0) to local on all nodes before parallel execution
    #pragma SMint parallel for copyin(vec1[ ]) // Parallel execution on all nodes
    #pragma omp parallel for
    for(i=0; i<N; i++) {
        for(k=0; k<N; k++)
            vec2[i]= array[i][k] * vec1[k]; // Matrix-vector multiplication
    }
}

```

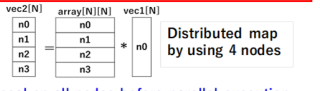


Fig. 4 SMint program with a *copyin* directive

III. PARALLEL PROGRAMMING IN XMP AND UPC

This section outlines the two PGAS languages, XcalableMP and UPC, which were used for comparison with SMint.

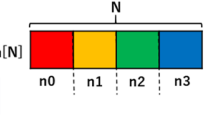
A. Overview of XcalableMP

XcalableMP (XMP) [4] is a directive-based PGAS language for distributed memory systems, providing two programming models; a global-view model and a local-view model. In global-view programming, data distribution mapping and process distribution mapping (work mapping) are specified as pragma directives. Fig. 5 shows examples of data and work mappings in XMP and SMint. They are similar programs but differ in execution. In XMP, the data access in a *for* statement is limited to local data, and a range of global array *a[N]* is allocated to the local node by a template *t[N]* specification. On the other hand, SMint has no restriction in the accessible range of *a[N]*. The underlying runtime system mSMS fetches the remote node data, if necessary, during the *for* loop execution.

```

...
#pragma xmp nodes p[4]
#pragma xmp template t[N]
#pragma xmp distribute t[block] onto p
double a[N];
#pragma xmp align a[i] with t[i]
...
int main(){
    int i;
    #pragma xmp loop on t[i] //work mapping
    for(i=0; i<N; i++)
        a[i]=... //
    return 0;
}

```

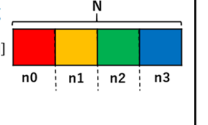


(a) XMP program

```

...
#pragma SMint shared ::[NPROCS](0, NPROCS)
double a[N];
...
int main(){
    int i;
    #pragma SMint parallel for //work mapping
    for(i=0; i<N; i++)
        a[i]=...
    return 0;
}

```



(b) SMint program

Fig. 5 Data mapping and work mapping in XMP and SMint

```

...
#pragma xmp shadow a[1][0] //Declaration of sleeve area
...
int main(){
    int i,j;
    #pragma xmp reflect(a) //Prefetch sleeve area
    #pragma xmp loop on t[i] //work mapping
    for(i=0; i<NY; i++)
        if(i==0 || i==NY-1) continue;
        for(j=1; j<NX-1; j++)
            b[i][j] = 0.4*a[i][j] + 0.15*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
    return 0;
}

```

(a) XMP program with *shadow* & *reflect* directives

```

...
int main(){
    int i;
    #pragma SMint parallel for scopyin(a[1][0]) //work mapping and prefetch sleeve area
    for(i=0; i<NY; i++)
        if(i==0 || i==NY-1) continue;
        for(j=1; j<NX-1; j++)
            b[i][j] = 0.4*a[i][j] + 0.15*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
    return 0;
}

```

(b) SMint program with a *scopy* directive

Fig. 6 Prefetch sleeve area in XMP and SMint

Fig. 6 shows description examples of typical stencil calculations for XMP and SMint. XMP, shown in Fig. 6(a), uses *shadow* and *reflect* constructs for prefetching sleeve areas. SMint, shown in Fig. 6(b), uses the *scopy* directive. Descriptions in both XMP and SMint increase the performance of their *parallel-for* execution by prefetching the data required in advance.

Fig. 7 shows descriptions of global data copy in XMP and SMint. In XMP, the *gmove* construct is necessary for global data copy, as shown in Fig. 7(a). In this example, *a[0]*–*a[9]* in node 0 are copied to *a[50]*–*a[59]* in node 2. A data block assignment statement is available in XMP. The decision for the *gmove* construct usage depends on whether *a[0]*–*a[9]* and *a[50]*–*a[59]* are mapped in the same node or in different nodes. In other words, it is always necessary to know the mapped location of

global data, although the array name “*a*” and index “*i*” can be used in the global namespace. In contrast, SMint provides seamless global data accessing independent of the location of global data. Therefore, SMint description is the same as that in a regular C program, as shown in Fig. 7(b). There data location can be ignored.

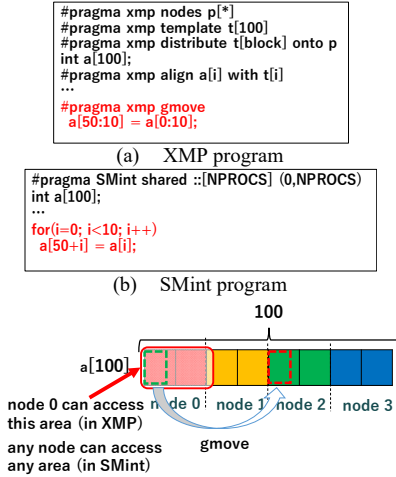


Fig. 7 Global data copy in XMP and SMint global-view model

In summary, global-view programming in XMP, which provides global data namespace and global data access, is only available in two cases: (1) access to sleeve areas by using *shadow* and *reflect* construct and (2) copy of global data by using *gmove* construct. This limitation in XMP reduces the variety of XMP applications in the global-view model.

On the other hand, the global view description in Fig. 7(a) can be rewritten to the local view description in Fig. 8. The global array *a*[100] is viewed as four arrays of *a*[25] in the XMP local-view model shown in Fig. 8. The local-view model uses “coarray notation”, which consists of a local array with local indexes followed by a “:” and the node ID number. In several PGAS languages, block data assignments using a coarray notation are directly converted to MPI one-sided communications.

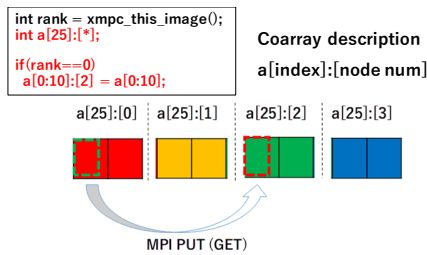


Fig. 8 Data copy using Coarray description in XMP local-view model

B. Overview of Berkeley-UPC

UPC [5,10] is one of the PGAS languages for distributed memory systems and Berkeley-UPC [7] is one of its implementations. In UPC, parallel processing is performed during an execution entity called THREAD, and THREADS is

the total number of THREAD(s). A UPC THREAD is not a thread; it is closer to a process. In UPC, it is possible to declare shared data in the global address space by adding the extended data type *shared* in front of the normal data declaration statements in C. Moreover, it is also possible to use pointers to point to data in the global address space. The array distributed mapping to THREADS is, by default, a single-element cyclic distribution, as shown in the first line of Fig. 9. By inserting the block size specification between *shared* and data type, as shown in the second line of Fig. 9, a block cyclic distributed mapping can be described.

As a result of the UPC language specification, it is impossible to distinguish between a local node THREAD and a remote node THREAD in UPC programs (the THREAD layout can be determined at compile time as `-pthread = “number of THREADS per node”` and at execution time by specifying the number of compute nodes). On the other hand, SMint and XMP provide two levels of parallel processing. Multi-node parallel and multi-core parallel in one node can be specified independently using an individual pragma statement, for example, `pragma SMint` and `pragma OMP`. Threads in the same compute node can access shared data efficiently without an overhead between remote nodes. In UPC, even where data are within the same node, access to the data of another THREAD (process) is obtained as global data access using a shared pointer, thus causing extra overhead.

Besides the existing library functions, UPC adds new syntax to the C language. In *upc_forall*, one of the new syntaxes, the fourth specification (affinity setting) is added to the *for* statement in C, which specifies the address of data the process will access. In Fig. 10, the THREAD holding *b*[*i*] executes iteration *i*.

```
shared double a[100]; //1 element cyclic
shared [100/THREADS] double b[100]; //block size is 100/THREADS
```

Fig. 9 Declaration of global data in UPC

```
//The process holding b[i] executes the iteration
upc_forall(i=0; i<100; i++; &b[i]){
    b[i]=...
}
```

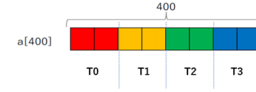
Fig. 10 A *upc_forall* statement in UPC

The global and local view programs in UPC are shown in Fig. 11. Both programs describe the same 1D, three-point stencil processing. In Fig. 11(a), processing for shared array *a*[400] is described in the global view model, whereby the program itself is very simple. However, the recommended version of programming, for improved performance, is based on a local-view model using local pointer *pa* and local index *i*, as shown in Fig. 11(b). In this program, local pointer *pa* is set to point to the first element of the subarray of *a*[400], which is mapped to the local THREAD. Using a local pointer *pa* pointing local subarray boosts the stencil calculation performance. Shared array *a* and global indexing is only used for the sleeve calculations. Thus, the main part of the program is a description of local array *pa*[100] using a local index (*i* = 0–99), which is no longer different from the MPI programming.


```
#include <stdio.h>
#include <upc.h>

shared [*] float a[400];
shared [*] float b[400];

int main(){
...
//The process holding a[i] executes the iteration
upc_forall(i=1; i<399; i++){
  b[i] = 0.3 * a[i-1] + 0.4 * a[i] + 0.3 * a[i+1];
}
}
```



(a) Global view model in UPC

```
#include <stdio.h>
#include <upc.h>

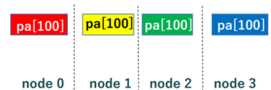
shared [*] float a[400];
shared [*] float b[400];

int main(){
int i;
float *pa = (float *) &a[400/THREADS * MYTHREADS];
float *pb = (float *) &b[400/THREADS * MYTHREADS];

if(MYTHREAD>0) //left sleeve
pb[0]=0.3*a[100*MYTHREAD-1] + 0.4*pa[0] + 0.3*pa[1];

for(i=1; i<99; i++){ //Each THREAD calculates pa[1]~pa[98]
pb[i] = 0.3 * pa[i-1] + 0.4 * pa[i] + 0.3 * pa[i+1];
}

if(MYTHREAD<THREADS-1) //right sleeve
pa[99]=0.3*pb[98] + 0.4*pb[99] + 0.3*b[100*(MYTHREAD+1)];
}
```



(b) Local view model in UPC

IV. THE PROGRAMMABILITY COMPARISON

A stencil calculation, one of the most typical and important calculation kernels in scientific computations, was used for the comparison in this study, because it is one of the few applications that can be implemented with the XMP global view model.

A. Data size limitations in UPC

In UPC, there is an upper limit on the block size of distributed mapping in the shared array declaration displayed in Fig. 9. The internal expression of the UPC shared pointer consists of three fields: block element size, number of THREADS, and memory address space per THREAD. The maximum available block size in the current version of UPC struct-based pointer [7] is only 31 bits = 2G elements. This means that the available mapped data size per THREAD is limited. Considering this limitation, two 2D data arrays with $65,536 \times 65,536$ elements were used for strong scaling performance measurements on multiple nodes larger than four nodes. Hence, the simplest five-point stencil calculation for comparatively small-size 2D arrays (64 GiB in total) was employed in this study.

B. SMint, XMP, and UPC programs in Global View Model

The 2D five-point stencil calculation was implemented in three languages and the programmability among them was compared. Figs. 12, 13, and 14 show the skeleton code of the stencil calculations implemented in each of the SMint, XMP, and UPC. Upon comparing SMint and XMP, we see that XMP requires more pragma directive statements than SMint. The UPC program in Fig. 14 (global pointer version) is based on global-view programming, which provides simple description and is easy to read. In contrast to the UPC program, SMint and

```
#include ...
#include <smint.h>
...
#pragma SMint shared ::[NPROCS][1] (0, NPROCS);
double a[NY][NX];
#pragma SMint shared ::[NPROCS][1] (0, NPROCS);
double b[NY][NX];

int main(int argc, char **argv){
int x,y,t;
//Array Initialize
...

for(t=0; t<NT; t++){ //time step
if(t%2==0) { //phase 1 : b=a
#pragma SMint parallel for scopyin(a[1][])
#pragma omp parallel for private(x)
for(y=0; y<NY; y++){
if(y==0 || y==NY-1) continue;
for(x=1; x<NX-1; x++){
b[y][x] = 0.4*a[y][x] + 0.15*(a[y-1][x] + a[y+1][x] + a[y][x-1] + a[y][x+1]);
}
}
}
else { //phase2 : a=b
#pragma SMint parallel for scopyin(b[1][])
...
}
} // time step end

return 0;
}
```

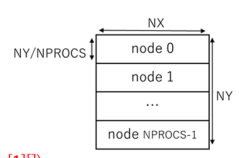


Fig. 12 A 2D five-point Stencil skeleton in SMint

```
#include ...
#include <xmp.h>
...
#pragma xmp nodes p[*] // Declaration of node set
#pragma xmp template t[NY] //Declaration of template (virtual array)
#pragma xmp distribute t[block] onto p
double a[NY][NX];
double b[NY][NX];
#pragma xmp align a[i][*] with t[i]
#pragma xmp align b[i][*] with t[i]
#pragma xmp shadow a[1][0] //Declaration of sleeve area
#pragma xmp shadow b[1][0]

int main(int argc, char **argv){ NY/xmp_num_nodes()
int i,j,t;
//Array Initialize
...

for(t=0; t<NT; t++){ //time step
if(t%2==0) { //phase 1 : b=a
#pragma xmp reflect (a) //Prefetch sleeve area of a
#pragma omp parallel for private(j)
#pragma xmp loop on t[i]
for(i=0; i<NY; i++){
if(i==0 || i==NY-1) continue;
for(j=1; j<NX-1; j++){
b[i][j] = 0.4*a[i][j] + 0.15*(a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
}
}
}
else { //phase2 : a=b
#pragma xmp reflect (b) //Prefetch sleeve area of b
...
}
} // time step end

#pragma xmp barrier
return 0;
}
```

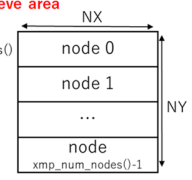


Fig. 13 A 2D five-point Stencil skeleton in XMP

XMP programs introduce two-level parallelism by adding an OpenMP pragma for multi-core parallelism in addition to multi-node parallelism.

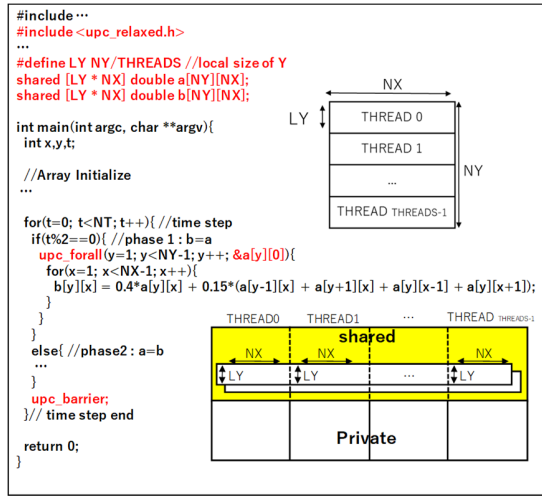


Fig. 14 A 2D five-point Stencil skeleton in UPC (global pointer version)

C. UPC Programs in Local View Model

To improve the performance of UPC programs, use of local view descriptions is recommended. In this section, two UPC programs using local view model are introduced alongside the global pointer version, as shown in Fig. 17. The performance of both the programs was evaluated through a comparison with the global-view program. The first one (local pointer version), shown in Fig. 15, uses a local pointer pa for the majority of the stencil calculations. The local pointer pa points to the first element of the range of the shared array a , mapped in the local node. The global pointer description using array a is used only for accessing the sleeve area in the upper and lower adjacent nodes, which are added separately beside the descriptions of calculation using pa .

The second program (*upc_alloc* version) is shown in Fig. 16. This program was developed with reference to the UPC programs in Parallel Research Kernel [15]. It uses a *upc_alloc* function instead of shared array declaration statements, to avoid the block size limitation in the shared data statements, as described previously. Array $a[NY][NX]$ is split into subarrays, $a[LY][NX]$, managed by each node (where $LY=NY/THREADS$), and allocated separately using *upc_alloc*. The program uses an array of the shared pointer pointing to a shared array, that is, $a[MYTHREAD]$, and a local pointer pointing shared data array, pa . The program uses the local pointer pa when accessing the local data in local THREAD, and it uses the shared pointer a when accessing remote data in other THREADS.

The local-view programs, shown in Figs. 15 and 16, are more complex compared to the global-view program, shown in Fig. 14. Many complicated pointer definitions and usages, shown in Fig. 16, hinder program readability and degrade program development productivity. It is difficult to determine conclusively whether local-view UPC programs provide a more efficient programming environment in comparison to the one MPI programs provide.

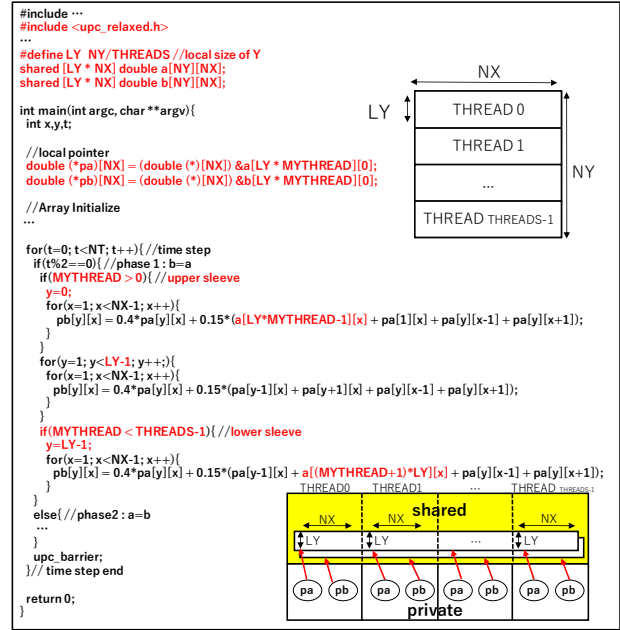


Fig. 15 A 2D five-point Stencil skeleton in UPC (local pointer version.)

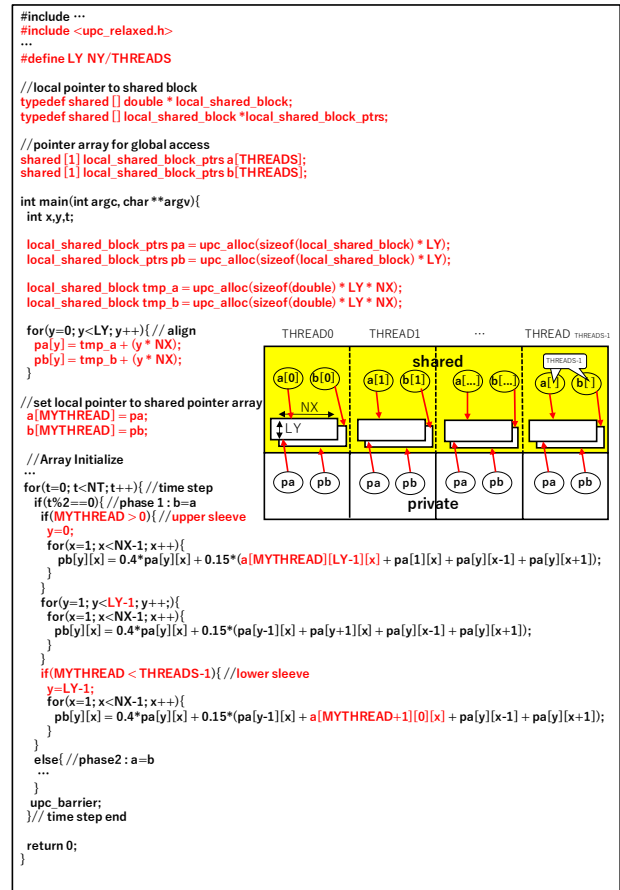


Fig. 16 A 2D five-point Stencil skeleton in UPC (*upc_alloc* version.)

V. THE PERFORMANCE COMPARISON

A. Experimental Environment

In this experiment, Omni XMP Compiler version 1.3.2 and Berkeley-UPC Compiler version 2.28.0 were used for XMP and UPC, individually. SMint and XMP used MPI, and UPC used GASNet's [8] MPI_conduit as the method of communication between nodes. The Tsubame 3.0 supercomputer, shown in Table 2, was used for performance measurements.

TABLE II. ENVIRONMENT ON TSUBAME3.0

CPU	Intel Xeon CPU E5-2680 v4 @ 2.40GHz * 2CPU
Num of Core / Threads	14 Core / 28 Threads
Memory	256GiB
Network	Intel Omni-Path HFI 100Gbps *4
OS	SUSE Linux Enterprise Server 12 SP2
Compiler	gcc 4.8.5
MPI	intel-mpi/18.1.163

B. Local and Global View Descriptions in UPC Programs

The performance of the three UPC descriptions, i.e., the *upc_alloc* version shown in Fig. 16, the global pointer version shown in Fig. 14, and the local pointer version shown in Fig 15, were investigated. Fig. 17 shows the execution times of the five-point stencil calculation (10 step iterations) carried out for a 2D array ($64K \times 64K$ elements, double) while using two computer nodes and 4–64 UPC THREADS in total. The execution time of the *upc_alloc* version was longer than those of the other two versions. While using four THREADS, the execution of the local pointer version was 13.1 times faster than that of the *upc_alloc* version, and 5.5 times faster than that of the global view version. The performance of all the versions of the UPC programs was accelerated by increasing the number of THREADS from 4 to 64.

C. Performance on a Various Number of Nodes and Threads

Fig. 18(a), (b), and (c) show the performance profiles of UPC, XMP, and SMint programs, respectively, while using various combinations of the number of nodes (2–16) and the number of threads per node (1–32).

Fig. 18(a) shows the execution times of the UPC local pointer version (which was depicted in Fig. 15). The best execution time was observed to be 10.7 s, achieved while using 2 nodes and 32 threads/node. In the case of the UPC local pointer version, the performance generally improved while using a small number of nodes and a large number of threads per node. On the other hand, the performance of UPC global pointer version improved while using a higher number of nodes and threads per node. Upon using more than two nodes, the performance of the global pointer version exceeded that of the local pointer version.

In Fig. 18(b), the best execution time recorded for XMP was 1.6 s when using 16 nodes and 8 threads per nodes. The XMP performance generally declined upon increasing the number of threads per node. In the case of the XMP, 8 or 16 were the optimal numbers of threads per node.

Fig. 18(c), shows the best execution time recorded in the case of SMint, i.e., 1.5 s while using 16 nodes and 32 threads per node. By increasing the number of nodes and threads per node,

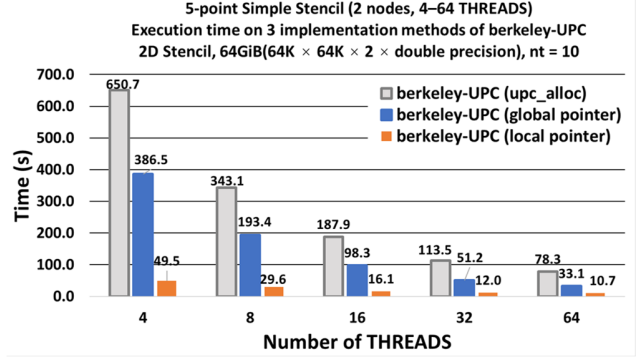
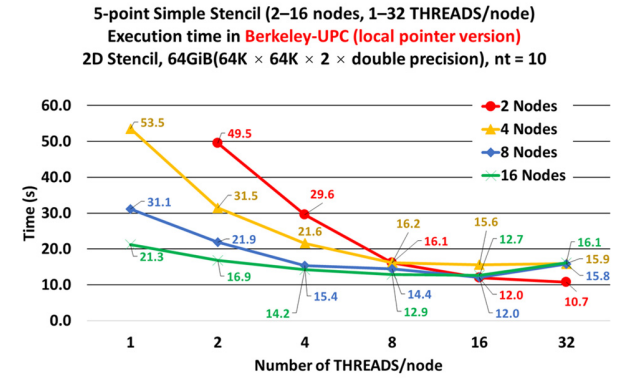
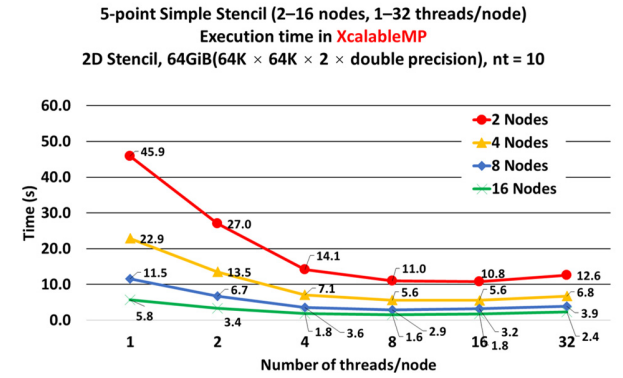


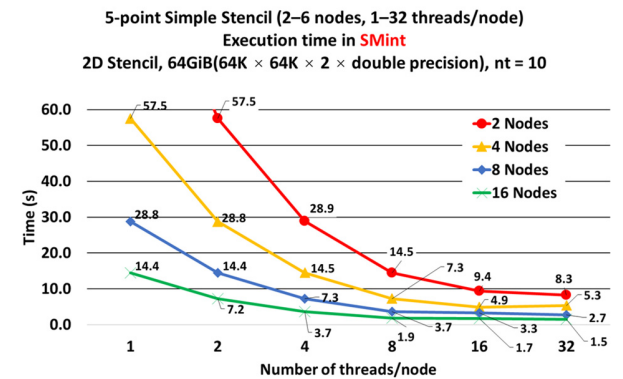
Fig. 17 Execution times on three implementation methods of UPC



(a) UPC (local pointer version)



(b) XMP



(c) SMint

Fig. 18 Execution times in three languages (2-16 nodes 1-32 threads/node)

the SMint program achieved a shorter execution time, which was a different tendency noticed when considering the other two programs (XMP and UPC local pointer versions).

D. Comparison of the Performance of the Three Languages

Fig. 19 shows a graph depicting the best execution times for each program upon using the specified number of nodes. In the cases where more than two nodes were used, the UPC execution times were 2.7–10.3 times slower than those of XMP and SMint. In contrast, upon comparing the results for SMint and XMP, the execution times obtained for SMint were 6–30% faster than those of XMP for any number of nodes.

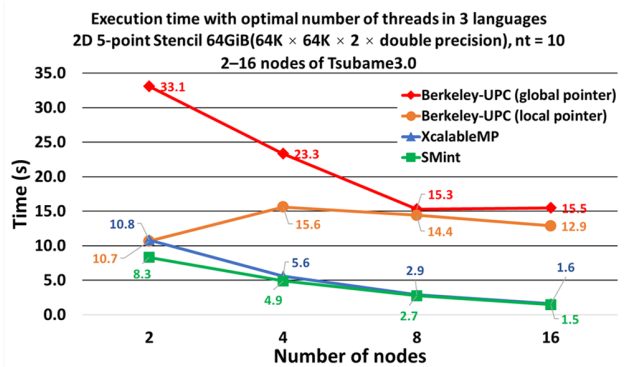


Fig. 19 Execution times with optimal number of threads in three languages

VI. CONCLUSION

In this study, the programmability and performance of global-view programming in SMint, XcalableMP, and UPC have been investigated using stencil computation. They are typical PGAS languages based on the C language and are available for multi-node and multi-core computations.

The global-view programming in XMP is only available in two cases: (1) access to sleeve areas using the *shadow* and *reflect* constructs and (2) copy of global data using the *gmove* construct. This limitation makes it difficult to use XMP for various applications. It is for this reason that stencil computation was employed for the global-view programming comparison in the study. The XMP performance in the stencil computation was approximately the same as the SMint performance.

In UPC, the global view program that utilizes shared data declaration statements is simple and supports high readability but performs poorly compared to XMP and SMint. Moreover, UPC global-view programs cannot be used for large-scale calculations since the global data array declaration cannot be used for large-scale data mapping when using several nodes. There is no alternative other than to employ the UPC local view programs using *upc_alloc*, local pointers, local arrays, and local array indexes. The UPC local-view descriptions usually have low readability and productivity because of the complicated pointer manipulations needed to access the local and shared data, as shown in Fig. 16.

In conclusion, there are several difficulties in programming in XMP and UPC when using a global-view model that supports

a genuine global address space. Moreover, it was found that there were limitations on applicable computations and the global data sizes that can be handled successfully.

SMint can access both global and local data seamlessly by using ordinary C pointers, and there are no restrictions on accessible global data areas. This is due to the underlying mSMS providing flexible remote data access capability at runtime. In mSMS, processes share a large virtual address space—that exceeds the size of the local physical memory run on each calculation node—and are executed in parallel. On the other hand, several PGAS languages usually employ static analysis and optimization by their compiler and a direct translation from the remote data access to MPI get/put communications. SMint on mSMS has a more dynamic mechanism and thus, it can be used for various applications.

ACKNOWLEDGMENT

This work was supported by the Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures and High Performance Computing Infrastructure in Japan (Project ID: jh190039-ISH) and JSPS KAKENHI (Grant Number JP18K11327).

REFERENCES

- [1] OpenMP, <https://www.openmp.org/> online 2019/7/22
- [2] OpenACC, <https://www.openacc.org/specification> online 2019/7/22
- [3] M. D. Wael, S. Marr, B. D. Fraine, T. V. Cutsem, and W. G. Meuter, "Partitioned global address space languages," *ACM Comput. Surv. (CSUR)*, vol. 47, July 2015.
- [4] XcalableMP, <http://www.xcalablemp.org/ja/> online 2019/7/22
- [5] UPC Consortium, UPC Language Specifications Version 1.3, <https://upc.lbl.gov/docs/user/upc-lang-spec-1.3.pdf>
- [6] R. Numwich and J. Reid, "Co-Array Fortran for parallel programming," Technical report ral-tr-1998-060, Rutherford Appleton Laboratory, August 1998.
- [7] Berkeley UPC, <http://upc.lbl.gov/>, ver.2.28.9 online 2019/7/22
- [8] GASNet, <https://gasnet.lbl.gov/> online 2019/7/22
- [9] Tsubame3, <http://www.gsic.titech.ac.jp/tsubame3> online 2019/7/22
- [10] T. E. Ghazawi, W. Carlson, T. Sterling, and K. Yelick, "UPC distributed shared memory programming," WILEY, 2005, ISBN-10-471-22048-5
- [11] H. Midorikawa, U. Ohashi, and H. Iizuka, "The design and implementation of user-level software distributed shared memory system: SMS - implicit binding entry consistency model," *IEEE Pacific Rim Conference on Communications Computers and Signal Processing*, pp. 299–302, 2001-08. (doi: 10.1109/PACRIM.2001.953582)
- [12] H. Midorikawa, "Stencil computations using software distributed shared memory mSMS on large-scale multicore nodes," *IPSI SIG Technical Reports*, vol. 2018-HPC-165, pp. 1–9, 2018.7.
- [13] H. Midorikawa, "The performance analysis of portable parallel programming interface MpC for SDSM and pthread," *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2005, IEEE/ACM CCGrid)*, vol. 2, pp. 889–896, 2005.5 (doi: 10.1109/CCGRID.2005.155865)
- [14] Y. Sakaguchi, K. Nishiya, and H. Midorikawa, "SMint: directive-based API for translating sequential programs to multi-node multi-core programs," *IPSI SIG Technical Reports*, vol. 2018-HPC-167, pp. 1–9, 2018.12.
- [15] Parallel Research Kernel github <https://github.com/jeffhammond/PRK> online 2019/7/22