2010 39th International Conference on Parallel Processing Workshops

Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems

Jinpil Lee
Graduate School of Systems
and Information Engineering
University of Tsukuba
Tsukuba, Japan
Email: jinpil@hpcs.cs.tsukuba.ac.jp

Mitsuhisa Sato
Center for Computational Sciences
University of Tsukuba
Tsukuba, Japan
Email: msato@hpcs.cs.tsukuba.ac.jp

***Abstract*—Although MPI is a de-facto standard for parallel programming on distributed memory systems, writing MPI programs is often a time-consuming and complicated process. XcalableMP is a language extension of C and Fortran for parallel programming on distributed memory systems that helps users to reduce those programming efforts. XcalableMP provides two programming models. The first one is the global view model, which supports typical parallelization based on the data and task parallel paradigm, and enables parallelizing the original sequential code using minimal modification with simple, OpenMP-like directives. The other one is the local view model, which allows using CAF-like expressions to describe inter-node communication. Users can even use MPI and OpenMP explicitly in our language to optimize performance explicitly. In this paper, we introduce XcalableMP, the implementation of the compiler, and the performance evaluation result. For the performance evaluation, we parallelized HPCC Benchmark in XcalableMP. It shows that users can describe the parallelization for distributed memory system with a small modification to the original sequential code.**

***Keywords*-High Performance Computing, Parallel Programming Language**

I. Introduction

Distributed memory systems such as PC clusters are typical platform for high performance computing, and most users write their programs using MPI(Message Passing Interface). Although MPI is a de-facto standard for parallel programming for distributed memory systems, writing MPI programs is often a cumbersome process because it forces users to describe data distributions and inter-node communication with primitive API functions.

On the other hand, OpenMP has been very successful in providing a simple parallel programming model for shared memory systems such as multi-core, many-core CPUs. In OpenMP's directive-based programming model, users add directives to their serial source code to describe data/task parallelism. This type of programming model allows incremental parallelization from a serial code with small programming effort.

The directive-based approach can be taken for distributed memory systems. XcalableMP[1], XMP in short, is a directive-based language extension which allows users to develop parallel programs for distributed memory systems easily and tune performance with minimal and simple notations.

In XMP, all kinds of parallelism including data distributions, inter-node communication should be written by users explicitly. XMP extends C and Fortran with OpenMP-like directives to describe data/task parallelism, inter-node communication and language extensions for one-sided communication. The explicit parallelism concept in XMP increases programming cost slightly. But users can control over the parallel execution and optimize their applications using XMP language features, or you can even use MPI functions with XMP.

In section II, we compare XMP with other parallel programming models for distributed memory systems. Section III shows the design concept of XMP and how to write parallel programs in our language. Section IV describes the implementation of the XMP compiler, and section V has performance measurements of the prototype compiler using HPC Challenge Benchmark. Section VI discusses future work related to hybrid parallel programming for SMP clusters, and we conclude the paper in section VII.

II. Related Work

There has been a lot of work to provide a simple, efficient parallel programming model for distributed memory systems. But none of them has become a de-facto standard except MPI.

The PGAS(Partitioned Global Address Space) model has become a hot topic recently. The PGAS model assumes a global shared memory space which can be referred by any threads, and a portion of it is local to each thread. PGAS shared memory space is distributed to each thread. Users can exploit the data locality to optimize their applications.

UPC(Unified Parallel C)[2] and CAF(Co-Array Fortran)[3] are major languages based on the PGAS model. UPC global shared memory hides inter-node communication under local memory access. While it reduces programming effort, the

1530-2016/10 $26.00 © 2010 IEEE
DOI 10.1109/ICPPW.2010.62

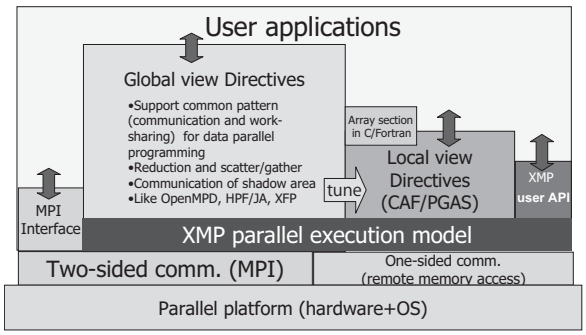

Fig. 1. Overview of XcalableMP API

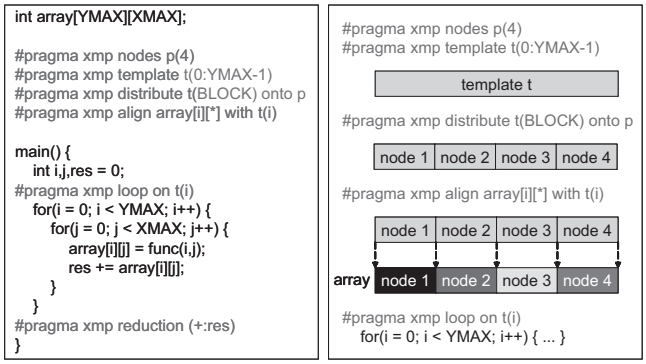

Fig. 2. Data Parallelization in Global View Model

performance is not always optimum. And it is often difficult to tune performance with the automatic inter-node communication.

CAF has a more explicit model. The reference of the global memory space should be done by using language extensions. The global memory is distributed to each process, and the user should use an extended assignment statement with target process's number which contains the data. Programming with CAF is very similar to MPI, in that it provides primitive functions for parallel programming. Since inter-node communication is clear to the users, performance tuning can easily be done. However, CAF requires a large amount of learning to code an efficient parallel program.

HPF(High Performance Fortran)[4] takes the directive-based approach similar to OpenMP and XMP. The user can describe data distribution using HPF directives. However, inter-node communication is automatically inserted by the compiler. This makes it difficult for the users to efficiently describe communication and optimize performance.

We proposed OpenMPD[5] which has a close programming model to XMP; it has directives to describe data parallelism and communication is explicit. But the language function is not enough for real applications, for example, it supports only 1-dimensional array distribution.

XMP has a directive-based language model. Many of the language features are inherited from the previous work, including HPF, CAF, OpenMPD and so on. Previous directive-based languages provide a simple programming model. But the performance is not always optimum, and difficult to tune it. To solve this problem, XMP takes totally explicit approach, which enables the user to control over the application's behavior and tune it.

III. Overview of XcalalbleMP

Figure 1 shows the overview of XcalableMP Application Program Interface. XcalableMP API is a collection of compiler directives, runtime library routines which can be used to describe data/task parallelism in C and Fortran programs. This specification provides a model of parallel programming for distributed memory systems.

XMP supports typical parallelization methods based on the data/task parallel paradigm under the "global view" model, and enables parallelizing the original sequential code using minimal modification with simple description, like OpenMP. It also includes CAF-like PGAS features as the "local view" model. Users can describe their own parallel algorithms and optimize their applications using more explicit and primitive language features for inter-node communication in the local view model. The important design principle of XMP is "performance-awareness". All actions for parallelization such as inter-node communication and work-sharing for loop are taken explicitly by user description; it is different from other automatic parallelizing programming models. The user should be aware of what happens by XMP description in the execution model on the distributed memory systems. This is very important for being "easy-to-understand" in performance tuning.

A. Execution Model

The basic execution model of XMP is the SPMD(Single Program Multiple Data) model, like MPI. As default, data declared in the program(without any XMP directives) is allocated on each node, and can only be referred by the allocated node. An XMP process begins its execution with a single thread on each node, which is equivalent to a single-threaded MPI process. Because of its design concept, explicit parallelism, memory access is always local, which means there is no automatic communication inserted by compilers. To access correct data in the parallel execution, users should synchronize the local buffer with inter-node communication. It can be described by XMP directives, and other language extensions.

B. Global View Programming Model

The global view programming model provides a simple way to describe a parallel program starting from a sequential version; the user parallelizes it by adding directives incrementally. Because these directives can be ignored as comments by sequential compilers of the base languages(C and Fortran), an XMP program derived from a sequential program can preserve the integrity of the original program when it is run sequentially.

Figure 2 shows a global view style code in XMP. The global view model shares major concepts with HPF. The programmer describes the data distribution of data shared among nodes by data distribution directives. The *node* directive declares a node

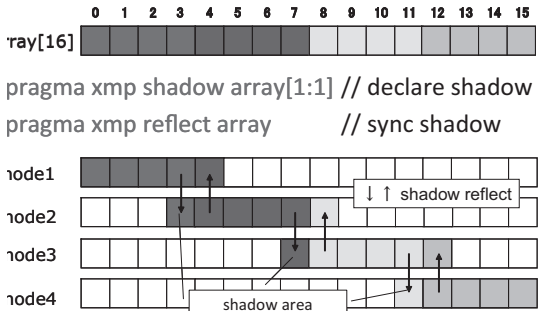

Fig. 3. Shadow Synchronization

set executing a XMP program, so the sample code would be executed on 4 nodes.

1) Data Distribution Using Templates: a template, a dummy array indicating data index space is declared(*template* directive) and distributed onto nodes(*distribute* directive). Block, cyclic, block-cyclic and gen-block distribution types are supported in the *distribute* directive. In the sample code, 1-dimensional template *t* is block distributed onto 4 nodes. Array distribution is declared by aligning the array to a template using the *align* directive. In the sample code, array *array*[*i*] is aligned to template *t*(*i*), that is, *array*[*i*] will be allocated on the owner node of *t*(*i*).

2) Work-Sharing: The *loop* directive splits up loop iterations among the executed nodes. The data accessed in a loop statement should be allocated in the local memory, because communication is explicit in XMP, that is, work-sharing and data distribution should be done in the same way. A template can be used in the *loop* directive to specify the data allocation. In the sample code, template *t* is used for parallelizing the loop statement. Consequently, the local part of the distributed array would be processed on each node.

When a function has no dependency with other ones, it can be executed in any order, and independent functions can even be executed in parallel. The *task* directive is used to execute a block of code on the specified node. For example, the following code execute the block statement on node *p*(1).

```
1  #pragma xmp task on p(1)
2  { ... code block ... }
```

The *task* directive describes task parallelization when independent tasks are executed on different nodes simultaneously.

3) Directives for Communication: The XMP compiler guarantees that communication takes place only when communication is explicitly specified. In global view model, communication directives are used to synchronize and keep the data consistency among the executed nodes.

When an array is distributed, the reference to the neighbor elements of the local block is a very typical access pattern causing inter-node communication. To access the neighbor elements, we need to extend the local block because all memory access is local in XMP. We call the extended area as a *shadow* of the array. Figure 3 shows the shadow area of array *array*. The *shadow* directive describes that the size(the number of elements) of shadow area on the array is 1 at both of the lower and upper side. A shadow is just a local memory buffer. To get the correct value of the neighbor elements, the data must be synchronized among the executed nodes. The *reflect* directive invokes inter-node communication copying the original data to the shadow area.

The *gmove* directive is a powerful operation in global view model; it copies a data block of a distributed array to another array in global view model. This directive is followed by an assignment statement with scalar variables, array references and array sections (In XMP, the C base language is extended to support array section notation). The assignment statement may require communication between nodes. The XMP compiler calculates element sets to be copied other nodes, and invoke proper communication. Figure 4 shows a sample code of the *gmove* directive. In this example, the first row of array *A*, *A*[0][:] is copied to array *L*. Without the *gmove* directive, the XMP compiler would try to copy the entire area of the first row of array *A* to array *L*. This will cause an error because array *A* is distributed onto 4 nodes. In this case, we should collect the array elements by communication. The compiler generates collective communication to collect the elements with the *gmove* directive; the owner node of *A*[*i*] sends the data to *L*[*i*] (This is a broadcast because array *L* is a local array).

Some typical type of collective communication can be implemented effectively. XMP provides communication directives for barrier, reduction and broadcast communication.

C. Local View Programming Model

The local view programming model provides language features describing remote memory access using one-sided communication extending the base languages. XMP adopts co-array notations compatible with CAF language [1] as an extension of the base languages. Figure 5 shows how to declare and use co-array in the C and Fortran version of XMP. Co-array *A* is declared using language extensions. Consequently,

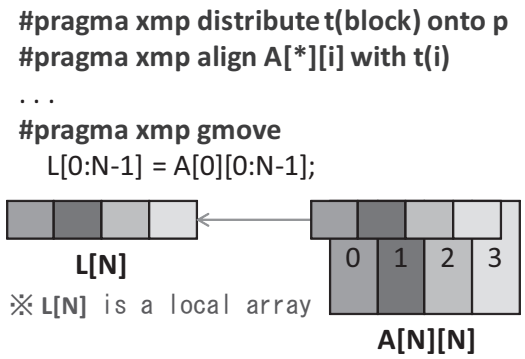

Fig. 4. Global Move Communication

[1]We extended C language to use co-array notations in XMP, but the syntax is slightly different with Fortran

```
<Fortran version>
real dimension A(N)[*]      ! declare co-array
B(1:N) = A(1:N)[1]          ! get A(1:N) from node1
<C version>
double A[N];
#pragma xmp coarray A[N]:(*)
B[0:N-1] = A[0:N-1]:(1);
```

Fig. 5. Co-Array in XcalableMP

an array of size N is allocated on each node, and extended array dimension from the base language called co-array dimension, which indicates a node set where the co-array is declared. Only co-array data can be remotely accessed in XMP. Extended assign statement specifying the target node number in co-array dimension is used for the remote memory access. In the sample code, every node gets the entire data of co-array *A* on node 1.

The local view model is suitable for the programs required to describe the parallel algorithm and remote data reference in more explicit way. As MPI is considered to have the local view of the data, the local view model of XMP has high interoperability with MPI.

IV. Compiler Implementation

We implemented a prototype compiler of XMP based on C language. Basic directives for data parallelism such as *template*, *align* and *loop*, and simple co-array notation which can be translated to get and put one-sided communication are supported in the prototype compiler. Figure 6 shows the translated code of the sample code in Figure 2. Because of its explicit parallelizing scheme, the compiler does not translate code without user descriptions. The compiler translates directives and other language extensions including co-array notations to runtime function calls. Distributed arrays are reallocated as 1-dimensional arrays in the constructor

```
nt **__array_addr;
__xmp_array_handle_t *__array_handle;                    // array descriptor

_xmp_main() {
  int i,j,res = 0, __local_i_lower, __local_i_upper;
  __local_i_lower = __xmp_get_lower(__array_handle, ...);  // calc lower bound
  __local_i_upper = __xmp_get_upper(__array_handle, ...);  // calc upper bound
  for(i = __local_i_lower; i < __local_i_upper; i++) {     // work-sharing
    for(j = 0; j < XMAX; j++) {
      *__XMP_GET_ADDR_2(__array_addr, i, j, ...) = func(i, j);
      res += *__XMP_GET_ADDR_2(__array_addr, i, j, ...);
    }
  }
  __xmp_allreduce(&res, ...);                              // reduction


__attribute__((constuctor)) static void __xmp_consturtor() {
// consturctor functions (nodes, templates, arrays, ...)
```

Fig. 6. Compiled Code of Figure 2

TABLE I
Node Configuration

CPU	AMD Opteron Quad-core 8000 series 2.3Ghz x 4 sockets (16 cores)
Memory	32GB
Network	Infiniband DDR (4 rails)
OS	Linux kernel 2.6.18 x86_64
MPI	MVAPICH2 1.2

section using GCC's __attribute__ extension. Reference to the arrays is translated to pointer reference of reallocated arrays. Information of array distribution is recorded in array, template descriptors.

Translated codes are passed to the native compiler(GCC) and linked with runtime library functions. Runtime library functions initialize and finalize the parallel environment, schedule loop iterations for work-sharing, invoke inter-node communication for data synchronization. Current implementation uses MPI for communication. Co-array notations describing one-sided communication are translated to *MPI_Get()*, *MPI_Put()* and *MPI_Accumulate()*.

V. Performance Evaluation

We used the HPCC(High Performance Computing Challenge) Benchmark[7] to evaluate the performance of the implemented compiler. HPCC Benchmark consists of 7 benchmarks, STREAM, RandomAccess, HPL, FFT, DGEMM, PTRANS and communication bandwidth/latency benchmark. We choose 4 benchmarks including STREAM, RandomAccess, HPL and FFT.

We used T2K-Tsukuba system[6] for evaluation. Figure I shows the node configuration of T2K-Tsukuba system. Each benchmark used 2, 4, 8, 16 and 32 physical system nodes. In this evaluation, we set up multiple XMP processes(equivalent to MPI processes) inside a physical node to exploit multicore architecture for STREAM and FFT. STREAM creates 15, HPL creates 16 processes per a physical node, and each process is assigned to a core. RandomAccess and FFT create one process on each node.

A. STREAM Benchmark

STREAM benchmark is a synthetic benchmark measuring memory bandwidth of target system. Vector a, b and c are accessed and modified by loop statements. Figure 7 shows the parallel code of STREAM Triad calculating $a = b + scalar \times c$. The parallelization of STREAM Triad is straightforward; it is a typical data-parallel program.

We describe the parallel version of STREAM Triad in the global view model of XMP. The *template* directive is written to declare data index space t of size *SIZE*, and distributed onto each node by block distribution. The *align* directive describes the vectors should be distributed aligned with the template t. The *loop* directive is added to describe that the owner of t(j) should execute the iteration j. To get total (triad) bandwidth of the system, reduce operation is invoked by the *reduction* directive after the calculation. The Lines Of Code, LOC in

```
double a[SIZE] , b[SIZE] , c[SIZE];
#pragma xmp nodes p(*)
#pragma xmp template t(0:SIZE−1)
#pragma xmp distribute t(BLOCK) onto p
#pragma xmp align [j] with t(j) :: a, b, c
. . .
# pragma xmp loop on t(j)
for (j = 0; j < SIZE; j++) a[j] = b[j] + scalar*c[j];
. . .
#pragma xmp reduction(+:triadGBs)
```

Fig. 7. Parallel Code of STREAM Triad

short, of STREAM Triad is 98 [2], and 12 lines(directives) of them are added to the sequential code to parallelize STREAM Triad.

Figure 8 shows the performance of STREAM Triad. STREAM Triad is embarrassingly parallel, every iteration can be executed in parallel and there is no communication during the vector access. Therefore, STREAM Triad shows good scalability related to the number of nodes.

B. RandomAccess Benchmark

RandomAccess benchmark measures the performance of random integer updates of memory. The measurement is Giga UPdates per Second(GUPS). Figure 9 shows the parallel code of RandomAccess. It updates arbitrary array elements for each iteration. The array update is written like the following statement in the sequential version.

```
1   Table[temp] ^= temp;
```

When an element to update is allocated on a remote node, the operation should be done using inter-node communication. Message passing with send/recv is not suitable for describing this kind of applications. The MPI version of RandomAccess creates a message handler processing remote update requests with long and complicated descriptions of MPI functions such as *MPI_Isend()*, *MPI_Irecv()* and *MPI_Test()*. Fundamentally, this is remote memory access. RandomAccess can be easily described with one-sided communication support.

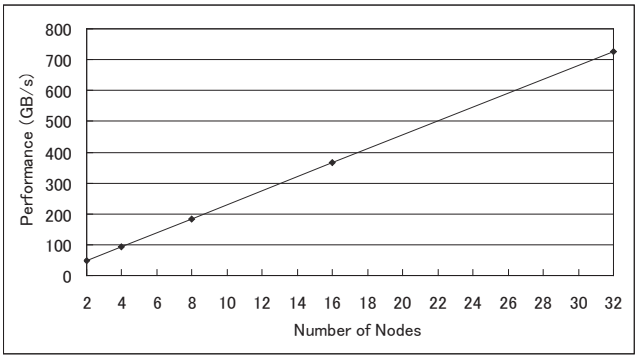

Fig. 8. Performance of STREAM Triad

[2]Each XMP directive is counted as one line.

```
#define SIZE TABLE_SIZE/PROCS
u64Int Table[SIZE] ;
#pragma xmp nodes p(PROCS)
#pragma xmp coarray Table [PROCS]
. . .
for (i = 0; i < SIZE; i++) Table[i] = b + i ;
. . .
for (i = 0; i < NUPDATE; i++) {
  temp = (temp << 1) ^ ((s64Int)temp < 0 ? POLY : 0);
  Table[temp%SIZE]:[(temp%TABLE_SIZE)/SIZE] ^= temp;
}
#pragma xmp barrier
```

Fig. 9. Parallel Code of RandomAccess

We parallelized RandomAccess in the local view model. In the local view model, data distribution is more explicit than the global view model. The updated array *Table* is divided by the number of nodes. We manually redefined the size of *Table* in the source code, and *SIZE* sized array is allocated on each node. Only the local *Table* can be accessed using local indices now. To enable remote memory access, we declared *Table* as a co-array by describing the *co-array* directive. RandomAccess create a random index *temp*(The range is from 0 to (TABLE_SIZE - 1)) on each iteration. The information for remote memory access is manually calculated and used in the co-array notation. ((temp%TABLE_SIZE)/SIZE) indicates the node number to access, and (temp%SIZE) indicates the local index of *temp* on the target node. Consequently, accumulation(BIT XOR) to *Table*[*temp*] can be done remotely. Remote memory access with co-array notation is asynchronous in XMP. Barrier synchronization is taken to complete requested remote memory access(*barrier* directive). This is for the performance evaluation. The LOC of RandomAccess is 77. We added 4 directives and modified 3 lines(array declaration and update) to the sequential code to parallelize RandomAccess.

Figure 10 shows the performance of RandomAccess. We couldn't achieve good performance on RandomAccess, and it shows bad scalability. The most of the overhead is barrier of remote memory access processing a large number of asynchronous messages. We are using MPI-2 functions such as *MPI_Get()*, *MPI_Put()*, *MPI_Accumulate()* and *MPI_Fence()* to implement one-sided communication; the co-array notations and barriers are translated those function calls. This benchmark shows the performance of remote memory access, and it tells we need more efficient implementation of one-sided communication.

C. Linpack Benchmark

Linpack benchmark measures the floating point rate of execution for solving a dense system of linear equations.

Figure 11 shows the parallel code of Linpack. We parallelized simple sequential Linpack routines(*dgefa* and *dgesl*) in global view model. Matrix a is distributed along the first dimension in a cyclic manner. Pivot vector *pvt_v* is an N sized local array duplicated on each node.

When a function processes distributed arrays(distributed by the global view directives), the function should also be parallelized. For example, *A_daxpy* in Figure 11 calculates $dy = dyda \times dx$. dx and dy are pointers referring arrays. When the referred arrays are distributed, the loop statement should be parallelized not to access unallocated area of the arrays. The *align* directive is often used in global scope to describe that the target (global) array is distributed. And the compiler creates the array descriptor and reallocates the target array. In a function's local scope, the *align* directive can be used for the function's parameters. It tells that the target parameter array is distributed in global scope (we assume that the target array is already distributed by the *align* directive in global scope), and creates the array descriptor which is used to parallelize the function. In *A_daxpy*, the descriptor is used to parallelize the loop. The size of parameter arrays should be written explicitly to create the array descriptor. Current prototype compiler only allows a constant number for the parameter size. We are fixing this to allow variable length arrays as parameters.

The main issue on parallelizing Linpack is exchanging the pivot vector. On iteration k, row l is selected as a pivot vector and row k is exchanged with row l. This is a typical operation of Gaussian elimination with partial pivoting. The problem is matrix a is distributed among nodes, that is, the row exchange requires inter-node communication. We used the *gmove* directive to describe the communication. The first *gmove* directive copies the l-th row to *pvt_v*. Then, the owner of row l broadcasts row data to *pvt_v*. The second *gmove* directive generates send/recv communication. The owner of row l receives row k from its owner. The third *gmove* directive is a local memory copy operation. The owner of row k copies *pvt_t* to row k. Consequently, row exchanging is completed using the pivot buffer. Those communications are generated by the compiler with the *gmove* directive descriptions, and users do not need to consider the owner node of each data.

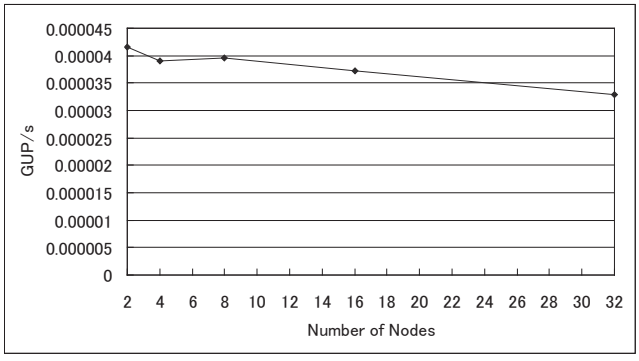

Fig. 10. Performance of RandomAccess

```
#pragma xmp nodes p(*)
#pragma xmp template t(0:N-1)
#pragma xmp distribute t(CYCLIC) onto p
double a[N][N], pvt_v[N];
#pragma xmp align a[*][i] with t(i)
. . .
void dgefa(double a[N][N], int n, int ipvt[N]) {
#pragma xmp align a[*][i] with t(i)
 . . .
 for (k = 0; k < nm1; k++) {
  . . .
#pragma xmp gmove
   pvt_v[k:n-1] = a[k:n-1][l];
   if (l != k) {
#pragma xmp gmove
    a[k:n-1][l] = a[k:n-1][k];
#pragma xmp gmove
    a[k:n-1][k] = pvt_v[k:n-1];
   }
   . . .
   for (j = kp1; j < n; j++) {
    t = pvt_v[j];
    A_daxpy(k+1, n-(k+1), t, a[k], a[j]);
   }
   . . .
}
void A_daxpy(int b, int n, double da, double dx [N], double dy[N]) {
#pragma xmp align [i] with t(i) :: dx, dy
 . . .
#pragma xmp loop on t(i)
 for (i = b; i < b+n; i ++)
  dy[i] = dy[i] + da*dx[i];
}
```

Fig. 11. Parallel Code of Linpack

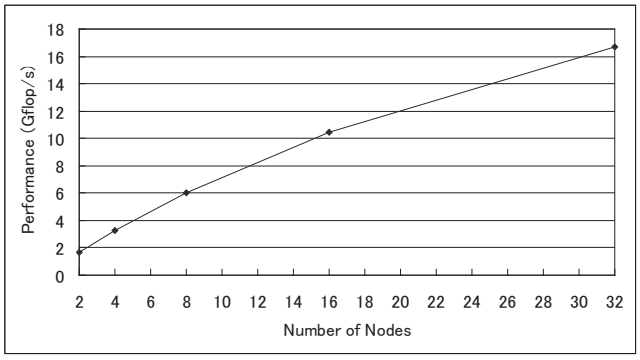

Fig. 12. Performance of Linpack

The LOC of Linpack is 243. We added 35 directives to the sequential code to parallelize.

Figure 12 shows the performance of Linpack. The performance is not satisfying. One of the reasons is its simple parallelization scheme, 1-dimensional array distribution. Another reason of bad scalability is communication in *dgesl* function, diagonal elements of matrix a should be broadcasted when

```
#pragma xmp nodes p(*)
#pragma xmp template t0(0:(N1*N2)-1)
#pragma xmp template t1(0:N1-1)
#pragma xmp template t2(0:N2-1)
#pragma xmp distribute (BLOCK) onto p :: t0, t1, t2
fftw_complex in[N1*N2], out[N1*N2], a_work[N2][N1];
#pragma xmp align [i] with t0(i) :: in, out
#pragma xmp align a_work[*][i] with t1(i)
. . .
int zfft1d(fftw_complex a[N], fftw_complex b[N], ...) {
#pragma xmp align [i] with t0(i) :: a, b
  . . . zfft1d0((fftw_complex **)a, (fftw_complex **)b, ...); . . .
}
. . .
void zfft1d0(fftw_complex a[N2][N1], fftw_complex b[N1][N2], ...) {
#pragma xmp align a[i][*] with t2(i)
#pragma xmp align b[i][*] with t1(i)
  . . .
#pragma xmp gmove
  a_work[:][:] = a[:][:];
#pragma xmp loop on t1(i)
  for (i = 0; i < N1; i++)
    for (j = 0; j < N2; j++)
      c_assgn(b[i][j], a_work[j][i]);
#pragma xmp loop on t1(i)
  for(i = 0; i < N1; i++) HPCC_fft235(b[i], work, w2, N2, ip2);
  . . .
}
```

Fig. 13. Parallel Code of FFT

solving b. Although the performance is not that good, Linpack can be described with a few directives and we can optimize the performance based on this parallel code. 2-dimensional array distribution and cache blocking will be the next step to achieve better performance.

D. FFT Benchmark

FFT benchmark measures the floating point rate of execution for double precision complex 1-dimensional Discrete Fourier Transform. Figure 13 shows the parallel code of FFT. The six-step FFT algorithm is used as in the MPI version. The main function invokes *zff1d* function. *in* and *out* are passed to a and b. Those 1-dimensional vectors are accessed as 2-dimensional matrices in *zff1d0* function. In *zff1d0* function, 1-dimensional FFT is performed on each dimension of the matrix. Therefore, matrix transpose operations are required during the calculation.

We parallelized *zff1d* and *zff1d0* function in global view model. Local *align* directives are describing distribution of parameter arrays. We distributed them along the second dimension in a block manner because FFT is done along the first dimension. Two templates of different sizes are used to distribute a and b. Because the block size is same($((N1 \times N2) \div p) = ((N2 \div p) \times N1) = ((N1 \div p) \times N2)$), the original 1-dimensional vectors can be accessed as 2-dimensional matrices in the parallel version. In six-step FFT, matrix transpose operation is done before 1-dimensional FFT.

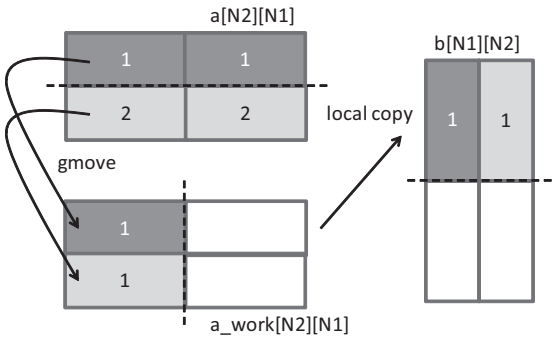

Fig. 14. Matrix Transpose in FFT

The matrix transpose is implemented by local memory copy between a and b in the sequential code. In the parallel version, the matrix transpose operation is implemented by the *gmove* directive and local memory copy. Figure 14 shows how matrix transpose a to b is processed on node 1. The number is the node number where the block is allocated, and dotted lines show how the matrices are distributed. Since the distribution manner is different, node 1 does not have all the elements of matrix a which are needed for the transpose. At first, a *gmove* is written to collect those elements. A new array *a_work* is declared to store the elements. *a_work* is distributed by *t1* which was used to distribute b. Consequently, the local block of *a_work* and b have the same shape. By the all-to-all communication of the *gmove* directive, all elements needed for transpose are stored in local buffer. So we can copy it to b using the loop statement [3]. And then, FFTE routine(FFT235) is used for 1-dimensional FFT. The rest of the code is simple work-sharing parallelizing loop statements. The LOC of FFT is 217. We added 31 directives to parallelize FFT.

Figure 15 shows the performance of FFT. The most of the overhead is all-to-all communication by the *gmove* directive. We have to improve *gmove* runtime functions get better performance. Or, users can describe overlapping gmove

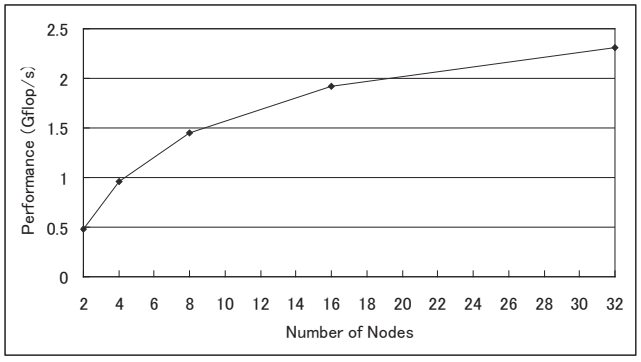

Fig. 15. Performance of FFT

[3] we cannot describe the transpose only by the *gmove* directive because of the syntax of the array section statement

communication with local copy using co-array functions.

VI. Future Work

One of the most important issues is to support multicore architecture, because a SMP cluster is now a standard platform in High Performance Computing area. SMP clusters have one or more multicore processor(s) sharing the local memory. To exploit the performance, we need to use all the calculating resources inside a node.

For example, MPI processes are parallelized in OpenMP(or more explicit way using thread libraries) which is called hybrid parallelization on SMP clusters. But hybrid programming of MPI and OpenMP requires high programming cost because MPI programming itself is a complicated job and the local view programming model has a weak affinity for thread-level parallelism. In this section, we discuss about language support for hybrid programming extending XMP. The language features introduced in this section is under design now.

Firstly, we consider the automatic thread-level parallelization. Figure 16 shows the image of hybrid parallelization in XMP. The left side of Pattern 1 shows a typical type of loop parallelization using the *loop* directive. We assume that each iteration is independent when the *loop* directive is described. This means the iterations are also executed independently in thread-level [4]. So we can translate the code to the right one. The compiler inserts OpenMP directives before the loop statement, which is already parallelized in XMP. Consequently, all cores would process different iteration sets in parallel.

XMP has the similar programming model with OpenMP, directive-based approach. So it has a strong affinity for OpenMP directives. It is not difficult to allow OpenMP directives in XMP source codes. We show an example in Pattern 2 of Figure 16. The user describes the OpenMP *parallel for* directive before the inner loop statement. And the XMP *loop* directive is described to parallelize the outer loop statement. This is a very typical example of hybrid programming. We can exploit parallelism using the hybrid memory architecture. The problem is explicit use of OpenMP directives cannot be combined with the automatic thread-level parallelization described in previous paragraph. To solve the problem, we introduce a new clause of the *loop* directive. *noOMP* describes that the target loop should not be parallelized in thread-level.

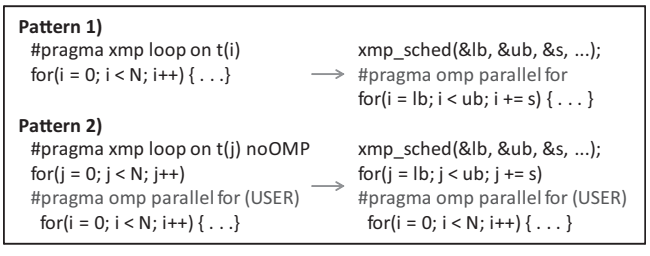

Fig. 16. Hybrid Parallelization for SMP clusters

[4]In the strict sense of the word, it is not automatic parallelization because we use the *loop* directive to detect thread-level parallelism.

VII. Conclusions

In this paper, we introduced a data parallel language extension for distributed memory systems named XcalableMP. XMP has two programming models. In global view model, various typical parallelization methods including array distribution, work sharing, and collective communication can be described with simple and easily understandable directives. In local view, one-sided communication is described using co-array notations. The performance evaluation using HPCC Benchmark shows that XMP achieves good performance for the typical type of data-parallel applications with small modifications to sequential codes. But at the same time, the result tells that one-sided communication should be enhanced. The current implementation is preliminary, and further enhancement is required. One of the interesting issues is hybrid programming on SMP clusters. We consider that the XMP compiler can generate OpenMP directives from the XMP directives.

Acknowledgment

The specification of XcalableMP has been being designed by the XcalableMP Specification Working Group which consists of members from academia, research labs and industries. This research is supported by "Seamless and Highly-productive Parallel Programming Environment for High-performance computing" project funded by Ministry of Education, Culture, Sports, Science and Technology, Japan.

References

[1] XcalableMP, http://www.xcalablemp.org/
[2] Unified Parallel C, http://upc.gwu.edu/
[3] Co-Array Fortran, http://www.co-array.org/
[4] High Performance Fortran, http://hpff.rice.edu/
[5] Jinpil Lee, Mitsuhisa Sato and Taisuke Boku, "OpenMPD: A Directive Based Data Parallel Language Extensions for Distributed Memory Systems", Proceedings of the 37th International Conference on Parallel Processing, pp.121-128, 2008.
[6] T2K Open Supercomputer, http://www.open-supercomputer.org/
[7] High Performance Computing Challenge Benchmark, http://icl.cs.utk.edu/hpcc/
[8] David Callahan and Ken Kennedy, "Compiling programs for distributed-memory multiprocessors", The Journal of Supercomputing vol.2, pp.151-169, 1988.
[9] Jingke Li and Marina Chen, "Compiling Communication-Efficient Programs for Massively Parallel Machines", IEEE Transactions on Parallel and Distributed Systems vol.2, pp.361-376, 1991.
[10] Chau-Wen Tseng, "An Optimizing FORTRAN D Compiler for MIMD Distributed Memory Machines", Ph.D Thesis, Rice University,1993.
[11] Seema Hiranandani, Ken Kennedy, John Mellor-Crummey and Ajay Sethi, "Compilation Techniques for Block-Cyclic Distributions", Proceedings of the 8th international conference on Supercomputing, pp.392-304, 1994.
[12] Charles Koelbel, "Compile-Time Generation of Regular Communications Patterns", Proceedings of the 1991 ACM/IEEE conference on Supercomputing, pp.101-110, 1991.
[13] Vikram Adve and John Mellor-Crummey, "Using Integer Sets for Data-Parallel Program Analysis and Optimization", Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, pp.186-198, 1998.
[14] Hidetoshi Iwashita, Masaki Aoki, "Mapping Normalization Technique on the HPF Compiler fhpf", Lecture Notes in Computer Science vol.4759, pp.315-328, 2009.
[15] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang and P. Sadayappan, "Compiling array expressions for efficient execution on distributed-memory machines", Journal of Parallel and Distributed Computing vol.32, pp.155-172, 1996.