SC09 HPC Challenge Submission for XcalableMP

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

Mitsuhisa Sato
Center for Computational Sciences
University of Tsukuba
msato@cs.tsukuba.ac.jp

I. Introduction

XcalableMP is a directive-based language extension which allows users to develop parallel programs for distributed memory systems easily and tune the performance by having minimal and simple notations.

Distributed Memory systems such as PC clusters are the typical platform for high performance computing, and most users write their programs using MPI. Although MPI is a de-facto standard for parallel programming for distributed memory systems, writing MPI programs is often a cumbersome and complicated process. So far, there have been a number of parallel programming languages for distributed memory architectures, including High Performance Fortran (HPF). However, these programming models have not been commonly used any more. On the other hand, OpenMP is widely used on shared memory architectures including SMP-configured PC clusters or multi-core CPUs. The most important feature of OpenMP from the perspective of programmability is to enable parallelization with simple directives that helps users extend their codes relatively easily from sequential ones. The target platform of OpenMP is however limited to shared memory architectures.

XcalableMP Application Program Interface (XcalableMP API) is a collection of compiler directives, runtime library routines that can be used to specify distributed-memory parallel programming in C and Fortran program. This specification provides a model of parallel programming for distributed memory multiprocessor systems, and the directives extend the C and Fortran base languages as to describe distributed memory parallel program, as in OpenMP. XcalableMP supports typical parallelization based on the data parallel paradigm and work mapping under ”global view”, and enables parallelizing the original sequential code using minimal modification with simple directives, like OpenMP. It also includes CAF-like PGAS (Partitioned Global Address Space) feature as ”local view” programming. The important design principle of XcalableMP is ”performance-awareness”. All actions of communication and synchronization are taken by directives, different from automatic parallelizing compilers. The user should be aware of what happens by XcalableMP directives in the execution model on the distributed memory architecture. This is very important for being “ easy-to-understand ” in performance tuning.

The specification has been being designed by XcalableMP Specification Working Group which consists of members from academia and research labs to industries. The development of prototype compilers are 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. XcalableMP is being designed based on the experiences of HPF, Fujitsu XPF (VPP Fortran) and OpenMPD. The XcalableMP specification and document are available at http:/www.xcalablemp.org.

In this submission we present results for the four HPC Challenge Type 2 programs - Stream, Random Access, FT and LU in XcalableMP using C as a base language. Since our compiler and runtime system of our implementation are still preliminary, we mainly focus on programmability of XcalableMP. We present performance numbers for Stream, Random Access and FT running on a Linux PC cluster.

II. XcalableMP Overview

A. Execution Model and Nodes directive

The target of XcalableMP is a distributed memory system. Each compute node, which may have several cores sharing main memory, has its own local memory, and each node is connected via network. Each node can access and modify its local memory directly, and can access the memory on the other nodes via communication. It is however assumed that accessing remote memory is much slower than the access of local memory.

The basic execution model of XcalableMP is a SPMD (Single Program Multiple Data) model on distributed memory. In each node, a program starts from the same main routine. An XcalableMP program begins as a single thread of execution in each node. The set of nodes when starting a program is called entire nodes.

When the thread encounters XcalableMP directives, the synchronization and communication occurs between nodes. That is, no synchronization and communications happen without directives. In this case, the program does duplicated execution

of the same program on local memory in each node. As default, data declared in the program is allocated in each node, and is referenced locally by threads executed in the node.

Node directive declares a node array to express a set of nodes. The following directives declare the entire nodes as an array of 16 nodes.

```
#pragma xmp nodes p(16)
```

A task is a specific instance of executable code and its data environments executed in a set of nodes. A task when starting a program in entire nodes is called an initial task. The initial task can generate a subtask which executes on a subset of the nodes by task construct. A set of nodes executing the same task is called executing nodes. If no task construct is encountered, a program is executed as one task, and its executing nodes are entire nodes.

The task construct is used to execute a block of code on the specified node. For example, the following code execute the block only on the master node (specified by 1), as master directive of OpenMP.

```
#pragma xmp task on 1
  { ···  block ··· }
```

XcalableMP supports two models of viewing data: global-view programming model and local-view programming model. In local-view programming model, accesses to data in remote nodes are done explicitly by language extension for get/put operations on remote nodes with node number of the target nodes, while reference to local data is executed implicitly.

B. Global View Programming Model

The global-view programming model is useful when, starting from sequential version of the program; the programmer parallelizes it in data-parallel model by adding directives incrementally with minimum modifications. As these directives can be ignored as a comment by the compilers of base languages (C and Fortran), an XcalableMP program derived from a sequential program can preserve the integrity of original program when it is run sequentially.

The global-view programming model shares major concepts with HPF. The programmer describes the data distribution of data shared among the nodes by data distribution directives. To specify the data distribution, the template is used as a dummy array distributed on nodes.

```
#pragma xmp nodes P(4)
#pragma xmp template T(0:15)
#pragma xmp distribute T(block) onto p
```

Block, cyclic, block-cyclic and gen-block distribution are supported. In this example, the one-dimensional template T is distributed on four nodes in the same size of blocks. A distributed array is declared by aligning the array to the template by align directive. In the following fragment of code, an array A is aligned to the template, that is, with block distribution

```
double A[16];
#pragma xmp align A[i] with T(i)
```

Figure 1 shows the assignment of a one-dimensional array A to four nodes. Each node is assigned an actual portion of the whole array to use, denoted by a gray/red colored part in Figure 1.

Figure 1. Data distribution and iteration

Loop construct maps iterations to the node where referenced data is located. Template is used to specify the mapping of iteration. By using the same template used for the data distribution, iterations are assigned to the node of the data. It should

be noted that in XcalableMP the programmer must control all data reference required computations done locally by any appropriate directives. For example, consider the following XcalableMP loop:

```
#pragma xmp loop on t(i)
  for(i = 2; i <= 10; i++)
    array[i] = . . .
```

Figure **??** shows an example where the loop actually scans the array elements a[2] to a[10] while the size of array is 16.

Global-view communication directives are used to synchronize between nodes, keep the consistency of shadow area, and move a part or all of distributed data globally. In XcalableMP, the inter-node communication must be described explicitly. The compiler guarantees that communication takes place only if communication is explicitly specified.

The gmove construct is a powerful operation in global-view programming in XcalableMP: It copies data of a distributed array in global-view. This directive is followed by the assignment statement of scalar value and array sections. The assignment operation of the array sections of a distributed array may require communication between nodes. In XcalableMP, C language is extended to support array section notation to support an assignment of array objects.

The gmove construct must be executed by nodes in the executing node set. And the value of scalar objects, and index value, range value of array section in the assignment statement must be same in every node executing this directive. When no option is specified, the copy operation is performed collectively by all nodes in the executing node set. In this case, all elements in both source array and target array must be distributed on to the executing node set. For example, the following example executes all-to-all communication to perform data copy between arrays which have different distribution.

```
double X[16][16], Y[16][16];
#pragma xmp align X[i][*] with T(i)
#pramga xmp align Y[*][i] with T(i)

. . .

#pragma xmp gmove
  X[:][:] = Y[:][:];  // array section assignment, which execute all-to-all comm
```

If the right hand side is owned by one node, the gmove operation is implemented as a broadcast communication.

The shadow directive specifies the shadow width of which area is used to communicate the neighbor element of block of a distributed array. The data stored in the storage area declared by the shadow directive is called shadow object. The reflect directive assigns the value of a reflection source to a shadow object for variables having the shadow attribute. Of the data allocated to a storage area other than a shadow area, data representing the same array element as that of a shadow object is called a reflection source of the shadow object.

For collective communications, barrier, reduction and broadcast operation are provided by the directives.

C. Local View Programming

Local view is suitable for the programs explicitly describing the algorithm of each node and explicit remote data reference. As MPI is considered to have the local view, the local view programming model of XcalableMP has high interoperability with MPI.

XcalableMP adopts coarray notations as an extension of languages for local view programming. In case of Fortran as the base language, most coarray notations are compatible to that of Coarray Fortran(CAF) expect that the task constructs are used for task parallelism. For example in Fortran, to access an array element of A(i) located on compute node N, the expression of A(i)[N] is used. If the access is a value reference, then the communication to get the value takes place. If the access is updating the value, then the communication to put a new value takes place.

In order to use coarray notations in C, we propose some language extension of the language. A coarray is declared by the coarray directive in C.

```
#pragma xmp coarray array-variable co-array-dimension
```

For example,

```
int A[10], B[10];
#pragma xmp coarray [*]: A, B
```

The coarray object is referenced in the following expression:

```
scalar-variable : [image-index]
array-section-expression:[image-index]
```

Array section notation is a notation to describe the part of array, which is adpted in Fortran90. In C, an array section has a form as follows:

```
array_name ‘ [‘  [lower_bound]‘ : ’[ upper_bound]  [‘ : ’ step] ‘ ] ’...
```

An array section is built from some subset of the elements of an array object - those associated with a selected subset of the index range attached to the object. The lower_bound and upper_bound specify the range of array elements of an array object. Either the lower bound or the upper bound can be omitted in the index range of a section, in which case they default to the lowest or highest values taken by the array's index. So A[:] is a section containing the whole of A. If the step is specified, the elements of an array section are every "step"-th element in the specified range. For example, B[1:10:3] is an array section of size 4 containing every third element of B with indices between 1 and 10 (ie, indices 1, 4, 7, 10). Collectively ranges specified by lower_bound, upper_bound and step are referred to as triplets. For multi-dimensional arrays, some dimensions could be subscripted with a normal scalar expression, and some could be "sectioned" with triplets.

For example,

```
A[:]= B[:]:[10]; // copy from B on image 10 to A
```

III. Implementation

The overall specification of XcalableMP API is still under design by the XcalableMP Specification Working Group. The current version of XcalableMP specification is 0.6. We implemented a part of XcalableMP API enough to implement the benchmark programs. The supported functions and limitations are described as follows:

1) Directives (global view model)
   The following directives are implemented in current version.
   - #pragma xmp nodes
   - #pragma xmp template
   - #pragma xmp distribute
     block, cyclic distribution can be discribed in distribute directive.
     block distribution with irregular chunk size is not supported yet.
     block-cyclic distribution is not supported yet.
   - #pragma xmp align
   - #pragma xmp loop
     loop can only be parallelized by template reference.
   - #pragma xmp task
     execution node on the task should be indicated by template.
   - #pragma xmp barrier
   - #pragma xmp reduction
     +, *, max, min, fistmax, lastmax operations are supported.
     Vector(data array) reduction is not supported yet.
   - #pragma xmp bcast
     Broadcast of vector is not supported yet.
   - #pragma xmp coarray
     multi-dimensional coarray is not supported yet.

   gmove directive used in Linpack and FFT is not implemented yet. We wrote communication functions in MPI, and called it explicitly in the benchmarks.
2) Remote Memory Access (local view model)
   Get/put for scalar variable can be used. The current compiler can parse array section in C language, but it cannot be used in coarray. In current version, coarray statement should be a simple assignment/accumulation. PLUS, logical XOR can be used in accumulation. Nested coarray expression cannot be used.
3) User API
   - int xmp_get_node_num(void);
     get node number of current communicator.
   - int xmp_get_num_nodes(void);
     get the number of nodes of current communicator.

- double xmp_get_second(void);
  get the current time(second)

The compiler was implemented by using Omni OpenMP compiler toolkit. It is a source-to-source compiler which translates a XcalableMP C program to C code with runtime libraries calls for communications. The runtime libraries are using MPI2 as a communication library.

IV. HPCC Results

A. Platform

We used typical Linux Cluster to evaluate the implemented benchmarks. Table.I shows the node configuration of the system. We used 16 nodes in maximum for the evaluation.

Table I
NODE CONFIGURATION

CPU	Intel(R) Core(TM)2 Quad CPU Q9650 @ 3.00GHz (x4)
Memory	8 GB
Network	1000 BASE-T Ethernet
OS	Linux kernel 2.6.28 x86_64
MPI	Open MPI 1.3.2

CLOC[2] by Al Danial is used to count Lines-Of-Code(LOC).

B. EP-STREAM-Triad

The program is quite straightforward. We describe parallelization of EP-STREAM-Triad in global view model of XcalableMP. In global view model, as in OpenMP, directives are used to parallelize base on the serial code.. "align " directive is used to distribute vectors onto each node. Each vector is distributed onto each node and processed simultaneously. "loop " directive is added to describe this work-mapping on loop. To get total (triad) bandwidth of the system, reduce operation is invoked by "reduction " directive.

1) Source Code and LOC: We show parallel code of EP-STREAM-Triad below. The LOC of the code is 98(each XcalableMP directive is counted as one line).

```
1 #include <stdio.h>
2 #include <float.h>
3 #include <math.h>
4
5 #define Mmin(a, b) (((a) < (b)) ? (a) : (b))
6
7 #define NTIMES 10
8 #define VECTOR_SIZE 134217728
9
10 #pragma xmp nodes p(*)
11 #pragma xmp template t(0:134217728-1)
12 #pragma xmp distribute t(block) onto p
13
14 int size, rank;
15 double a[VECTOR_SIZE], b[VECTOR_SIZE], c[VECTOR_SIZE];
16
17 int
18 checkSTREAMresults(void)
19 {
20   int j, k;
21   double aj, bj, cj, scalar, asum, bsum, csum, epsilon;
22
23   aj = 2.0;
24   bj = 2.0;
25   cj = 0.0;
```

```
26   scalar = 3.0;
27
28   for (k = 0; k < NTIMES; k++) aj = bj + scalar*cj;
29
30   aj = aj * (double) VECTOR_SIZE;
31   bj = bj * (double) VECTOR_SIZE;
32   cj = cj * (double) VECTOR_SIZE;
33
34   asum = 0.0;
35   bsum = 0.0;
36   csum = 0.0;
37   epsilon = 1.e-8;
38
39 #pragma xmp loop on t(j) reduction(+:asum, bsum, csum)
40   for (j = 0; j < VECTOR_SIZE; j++) {
41     asum += a[j];
42     bsum += b[j];
43     csum += c[j];
44   }
45
46   if (fabs(aj-asum)/asum > epsilon) {
47     printf("[%d] Failed Validation on array a[]\n", rank);
48     return 1;
49   }
50   else if (fabs(bj-bsum)/bsum > epsilon) {
51     printf("[%d] Failed Validation on array b[]\n", rank);
52     return 1;
53   }
54   else if (fabs(cj-csum)/csum > epsilon) {
55     printf("[%d] Failed Validation on array c[]\n", rank);
56     return 1;
57   }
58   else {
59     printf("[%d] Solution Validates\n", rank);
60     return 0;
61   }
62 }
63
64 int
65 HPCC_Stream(double *triadGBs)
66 {
67   register int j, k;
68   double scalar, times[NTIMES], mintime = FLT_MAX, curGBs;
69
70 #pragma xmp loop on t(j)
71   for (j = 0; j < VECTOR_SIZE; j++) {
72     a[j] = 1.0;
73     b[j] = 2.0;
74     c[j] = 0.0;
75   }
76
77 #pragma xmp loop on t(j)
78   for (j = 0; j < VECTOR_SIZE; j++) a[j] = 2.0E0 * a[j];
79
```

```
 80   scalar = 3.0;
 81   for (k = 0; k < NTIMES; k++) {
 82     times[k] = -xmp_get_second();
 83 #pragma xmp loop on t(j)
 84     for (j = 0; j < VECTOR_SIZE; j++) a[j] = b[j] + scalar*c[j];
 85     times[k] += xmp_get_second();
 86   }
 87
 88   for (k = 1; k < NTIMES; k++) {
 89     mintime = Mmin(mintime, times[k]);
 90   }
 91
 92   curGBs = (mintime > 0.0 ? 1.0 / mintime : -1.0);
 93   curGBs *= 1e-9 * 3 * sizeof(double) * (VECTOR_SIZE / size);
 94   *triadGBs = curGBs;
 95
 96   return checkSTREAMresults();
 97 }
 98
 99 int
100 main(void)
101 {
102   int err;
103   double triadGBs;
104
105   rank = xmp_get_node_num();
106   size = xmp_get_num_nodes();
107
108   err = HPCC_Stream(&triadGBs);
109 #pragma xmp reduction(+:err)
110 #pragma xmp reduction(+:triadGBs)
111
112   if (rank == 0) {
113     printf("%d Errors Found on %d nodes\n", err, size);
114     printf("Total Triad GB/s %.6f\n", triadGBs);
115   }
116
117   return 0;
118 }
```

2) Performance Archieved: We shows performance evaluation results in Figure 2. The vector size is 134217728. Because the benchmark is embarrassingly parallel, the performance scales well.

C. RandomAccess

The Random Access program is also straightforward. Random Access updates arbitrary array elements for each iteration. We parallelize this program in local-view programming model. XcalableMP extends the C language to allow coarray notation[3]. Remote memory access is used to update the element by coarray mechanism. Line 18 declare the array Table as a coarray, and distribute data manually onto each nodes (Line 79). Coarray is one of the important features of local view model of XcalableMP. Using coarray, users can describe remote memory access. For example, accumulate remote memory access (BIT XOR) is described using coarray at Line 63. "barrier " directive blocks until all processes of current communicator have reached it. But it can be also used to synchronize coarray. At Line 65, barrier synchronization is taken to complete the remote memory access.

1) Source Code and LOC: We show parallel code of RandomAccess below. The LOC of the code is 77.

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

node(s)	GB/s
1	4.17031
2	8.34807
4	16.6900
8	32.6532
16	65.5310

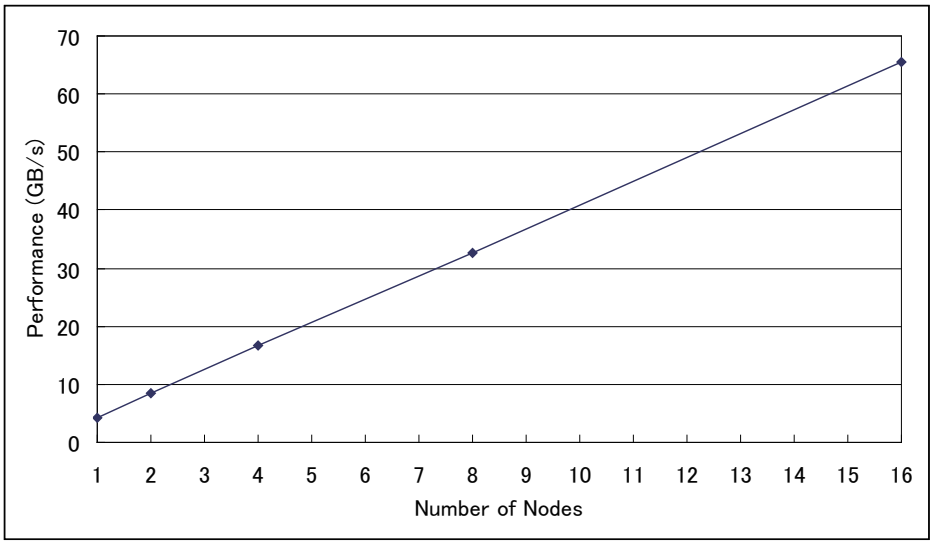

Figure 2. EP-STREAM-Triad Performance

```
#include <stdlib.h>

typedef unsigned long long      u64Int;
typedef signed long long        s64Int;

#define POLY                    0x0000000000000007ULL
#define PERIOD                  1317624576693539401LL
#define NUPDATE                 (4 * XMP_TABLE_SIZE)

#define XMP_TABLE_SIZE          131072
#define PROCS                   2
#define LOCAL_SIZE              XMP_TABLE_SIZE/PROCS

u64Int Table[LOCAL_SIZE];

#pragma xmp nodes p(*)
#pragma xmp coarray Table[*]

u64Int
HPCC_starts(s64Int n)
{
  int i, j;
  u64Int m2[64];
  u64Int temp, ran;

  while(n < 0) n += PERIOD;
  while(n > PERIOD) n -= PERIOD;
  if(n == 0) return 0x1;

  temp = 0x1;
  for(i = 0; i < 64; i++) {
    m2[i] = temp;
```

```
34      temp = (temp << 1) ^ ((s64Int) temp < 0 ? POLY : 0);
35      temp = (temp << 1) ^ ((s64Int) temp < 0 ? POLY : 0);
36    }
37
38    for(i = 62; i >= 0; i--)
39      if((n >> i) & 1) break;
40
41    ran = 0x2;
42    while(i > 0) {
43      temp = 0;
44      for(j = 0; j < 64; j++)
45        if((ran >> j) & 1) temp ^= m2[j];
46      ran = temp;
47      i -= 1;
48      if((n >> i) & 1)
49        ran = (ran << 1) ^ ((s64Int) ran < 0 ? POLY : 0);
50    }
51
52    return ran;
53  }
54
55  static void
56  RandomAccessUpdate(u64Int s)
57  {
58    u64Int i, temp;
59
60    temp = s;
61    for(i = 0; i < NUPDATE/128; i++) {
62        temp = (temp << 1) ^ ((s64Int) temp < 0 ? POLY : 0);
63        Table[temp%LOCAL_SIZE]:[(temp%XMP_TABLE_SIZE)/LOCAL_SIZE] ^= temp;
64    }
65  #pragma xmp barrier
66  }
67
68  int
69  main(void)
70  {
71    int rank, size;
72    u64Int i, b, s;
73    double time, GUPs;
74
75    rank = xmp_get_node_num();
76    size = xmp_get_num_nodes();
77    b = (u64Int)rank * LOCAL_SIZE;
78
79    for(i = 0; i < LOCAL_SIZE; i++) Table[i] = b + i;
80    s = HPCC_starts((s64Int)rank);
81
82    time = -xmp_get_second();
83    RandomAccessUpdate(s);
84    time += xmp_get_second();
85
86    GUPs = (time > 0.0 ? 1.0 / time : -1.0);
87    GUPs *= 1e-9*NUPDATE;
```

```
88
89 #pragma xmp reduction(+:GUPs)
90
91   if(rank == 0) {
92     printf("Executed on %d node(s)\n", size);
93     printf("Time used: %.6f seconds\n", time);
94     printf("%.9f Billion(10^9) updates per second [GUP/s]\n", GUPs);
95   }
96
97   return 0;
98 }
```

2) Performance Archieved: We shows performance evaluation results in Figure 3. The size of table to be updated is 131072. This results shows the performance of remote memory access in XcalableMP.

node(s)	GUP/s
1	0.02305
2	0.02154
4	0.02289
8	0.03597
16	0.03369

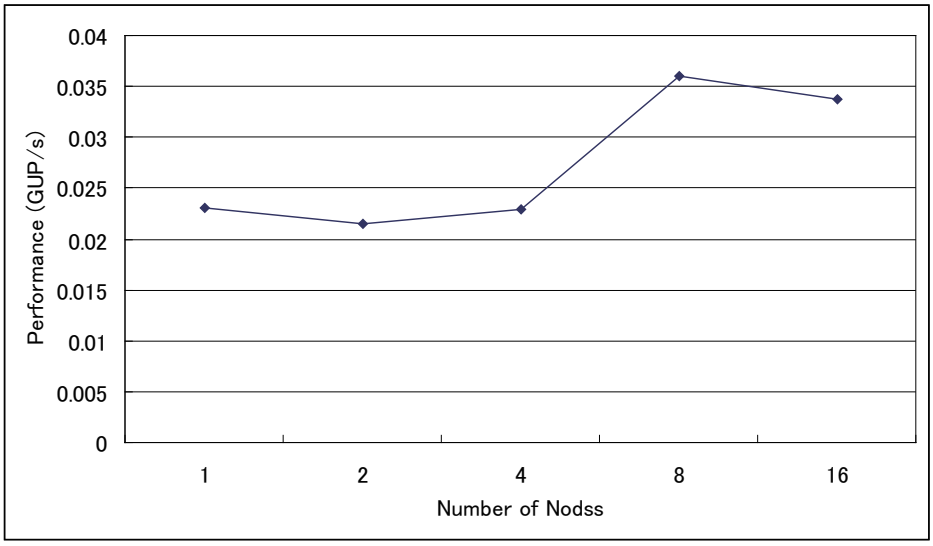

Figure 3. RandomAccess Performance

D. Linpack

We parallelized the simple sequential Linpack routine (dgefa and dgesl) in Global-view programming model. The matrix is distributed in a cyclic manner. The pivot is broadcasted and exchanged by gmove directive. The pivot is stored locally in each node. The BLAS routine is rewritten to express the distribution of the vector, while the original sequential BLAS takes only the first element of the partial array as a pointer. The parallelization is quite straightforward. Note that this implementation is not optimal one, but it shows how easy to parallelize a regular numerical algorithm in global-view programming mode of XcalalbleMP. The dmpxy used in verifications is implemented as an external routine due to some compiler problem.

1) Source Code and LOC: We show parallel code of Linpack below. The LOC of the code is 243.

```
1 #include <math.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 #define ZERO 0.0
6 #define ONE 1.0
7 #define EPS 1.0e-8
8
9 #define N 1024*8
10
```

```
11 double a[N][N], b[N], x[N], pvt_v[N];
12 int ipvt[N], n, rank, size;
13
14 #pragma xmp nodes p(*)
15 #pragma xmp template t(0:(1024*8)-1)
16 #pragma xmp distribute t(cyclic) onto p
17 #pragma xmp align a[*][i] with t(i)
18 #pragma xmp align b[i] with t(i)
19 #pragma xmp align x[i] with t(i)
20
21 void
22 matgen(double a[N][N], int n, double b[N] , double *norma)
23 {
24   int init, i, j;
25   double norma_temp;
26 #pragma xmp align a[*][i] with t(i)
27 #pragma xmp align b[i] with t(i)
28
29   srand48(rank);
30   *norma = 0.0;
31
32   for (j = 0; j < n; j++) {
33 #pragma xmp loop on t(i)
34     for (i = 0; i < n; i++) {
35       a[j][i] = drand48();
36       norma_temp = (a[j][i] > norma_temp) ? a[j][i] : norma_temp;
37     }
38   }
39 #pragma xmp reduction(max:norma_temp)
40
41   *norma = norma_temp;
42
43 #pragma xmp loop on t(i)
44   for (i = 0; i < n; i++) {
45     b[i] = 0.0;
46   }
47
48   for (j = 0; j < n; j++) {
49 #pragma xmp loop on t(i)
50     for (i = 0; i < n; i++) {
51       b[i] = b[i] + a[j][i];
52     }
53   }
54 }
55
56 int
57 A_idamax(int b, int n, double dx[N])
58 {
59   double dmax, g_dmax, temp;
60   int i, ix, itemp;
61 #pragma xmp align dx[i] with t(i)
62
63   if (n < 1) return -1;
64   if (n == 1) return 0;
```

```
65
66    itemp = 0;
67    dmax = 0.0;
68
69 #pragma xmp loop on t(i) reduction(lastmax:dmax/itemp/)
70    for (i = b; i < b+n; i++) {
71      temp = dx[i];
72      if(fabs(temp) >= dmax) {
73        itemp = i;
74        dmax = fabs(temp);
75      }
76    }
77
78    return itemp;
79 }
80
81 void
82 A_dscal(int b, int n, double da, double dx[N])
83 {
84    int i, m, mp1, nincx;
85 #pragma xmp align dx[i] with t(i)
86
87    if (n <= 0) return;
88
89 #pragma xmp loop on t(i)
90    for (i = b; i < b+n; i++)
91      dx[i] = da*dx[i];
92 }
93
94 void
95 A_daxpy(int b, int n, double da, double dx[N], double dy[N])
96 {
97    int i;
98 #pragma xmp align dx[i] with t(i)
99 #pragma xmp align dy[i] with t(i)
100
101   if (n <= 0) return;
102   if (da == ZERO) return;
103
104 #pragma xmp loop on t(i)
105   for (i = b; i < b+n; i++) {
106     dy[i] = dy[i] + da*dx[i];
107   }
108 }
109
110 void
111 dgefa(double a[N][N], int n, int ipvt[N])
112 {
113   double t;
114   int j, k, kp1, l, nm1, i;
115   double x_pvt;
116 #pragma xmp align a[*][i] with t(i)
117
118   nm1 = n-1;
```

```
119
120   for (k = 0; k < nm1; k++) {
121
122     kp1 = k+1;
123     l = A_idamax(k, n-k, a[k]);
124     ipvt[k] = l;
125
126 #pragma xmp task on t(l)
127     if(a[k][l] == ZERO) {
128       printf("ZERO is detected\n");
129       exit(1);
130     }
131
132 #pragma xmp gmove
133     pvt_v[k:n-1] = a[k:n-1][l];
134
135     if (l != k) {
136 #pragma xmp gmove
137       a[k:n-1][l] = a[k:n-1][k];
138 #pragma xmp gmove
139       a[k:n-1][k] = pvt_v[k:n-1];
140     }
141
142     t = -ONE/pvt_v[k];
143     A_dscal(k+1, n-(k+1), t, a[k]);
144
145     for (j = kp1; j < n; j++) {
146       t = pvt_v[j];
147       A_daxpy(k+1, n-(k+1), t, a[k], a[j]);
148     }
149   }
150
151   ipvt[n-1] = n-1;
152 }
153
154 void
155 dgesl(double a[N][N], int n, int ipvt[N], double b[N])
156 {
157   double t;
158   int k, kb, l, nm1;
159 #pragma xmp align a[*][i] with t(i)
160 #pragma xmp align b[i] with t(i)
161
162   nm1 = n-1;
163
164   for (k = 0; k < nm1; k++) {
165
166     l = ipvt[k];
167 #pragma xmp gmove
168     t = b[l];
169
170     if (l != k) {
171 #pragma xmp gmove
172       b[l] = b[k];
```

```
173 #pragma xmp gmove
174       b[k] = t;
175     }
176 
177     A_daxpy(k+1, n-(k+1), t, a[k], b);
178   }
179 
180   for (kb = 0; kb < n; kb++) {
181     k = n - (kb+1);
182 #pragma xmp task on t(k)
183 {
184     b[k] = b[k]/a[k][k];
185     t = -b[k];
186 }
187 #pragma xmp bcast t from t(k)
188 
189     A_daxpy(0, k, t, a[k], b);
190   }
191 }
192 
193 double
194 epslon(double x)
195 {
196   double a, b, c, eps;
197   a = 4.0e0/3.0e0;
198   eps = ZERO;
199 
200   while (eps == ZERO) {
201     b = a - ONE;
202     c = b + b + b;
203     eps = fabs(c-ONE);
204   }
205 
206   return(eps*fabs(x));
207 }
208 
209 double buffer[N];
210 
211 void
212 dmxpy(int n, double y[N], double *x, double m[N][N])
213 {
214   int i, j;
215   double temp;
216 #pragma xmp align m[*][i] with t(i)
217 #pragma xmp align y[i] with t(i)
218 
219 #pragma xmp gmove
220   buffer[:] = x[:];
221 
222 #pragma xmp loop on t(i)
223   for (i = 0; i < n; i++) {
224     temp = 0;
225     for (j = 0; j < n; j++) {
226       temp = temp + m[j][i]*buffer[j];
```

```
227     }
228     y[i] = y[i] + temp;
229   }
230 }
231
232 void
233 check_sol(double a[N][N], int n, double b[N], double x[N])
234 {
235   int i;
236   double norma, normx, residn, resid, eps, temp_b, temp_x;
237 #pragma xmp align a[*][i] with t(i)
238 #pragma xmp align b[i] with t(i)
239 #pragma xmp align x[i] with t(i)
240
241 #pragma xmp loop on t(i)
242   for (i = 0; i < n; i++) {
243     x[i] = b[i];
244   }
245
246   matgen(a, n, b, &norma);
247
248 #pragma xmp loop on t(i)
249   for (i = 0; i < n; i++) {
250     b[i] = -b[i];
251   }
252
253   dmxpy(n, b, x, a);
254
255   resid = 0.0;
256   normx = 0.0;
257
258 #pragma xmp loop on t(i) reduction(max:resid, normx)
259   for (i = 0; i < n; i++) {
260     temp_b = b[i];
261     temp_x = x[i];
262     resid = (resid > fabs(temp_b)) ? resid : fabs(temp_b);
263     normx = (normx > fabs(temp_x)) ? normx : fabs(temp_x);
264   }
265
266   eps = epslon((double)ONE);
267   residn = resid/(n*norma*normx*eps);
268
269   if(rank == 0) {
270     printf("   norm. resid      resid           machep\n");
271     printf("%8.1f      %16.8e%16.8e\n",
272            residn, resid, eps);
273   }
274 }
275
276 int
277 main(void)
278 {
279   int i,j;
280   double ops,norma, t0,t1, dn;
```

```
281
282   rank = xmp_get_node_num();
283   size = xmp_get_num_nodes();
284
285   if(rank == 0)
286     printf("Linpack ...\n");
287
288   n = N;
289   dn = N;
290   ops = (2.0e0*(dn*dn*dn))/3.0 + 2.0*(dn*dn);
291
292   matgen(a, n, b, &norma);
293
294   t0 = xmp_get_second();
295   dgefa(a, n, ipvt);
296   dgesl(a, n, ipvt, b);
297   t1 = xmp_get_second();
298
299   if(rank == 0)
300     printf("time=%g, %g MFlops\n",t1-t0, ops/((t1-t0)*1.0e6));
301
302   check_sol(a, n, b, x);
303
304   if(rank == 0)
305     printf("end ...\n");
306
307   return 0;
308 }
```

2) Performance Archieved: We shows performance evaluation results in Figure 4. The matrix size is 67108864 ((8x1024) x (8x1024)).

node(s)	MFlops
1	319.457
2	361.099
4	640.907
8	1107.05
16	1632.01

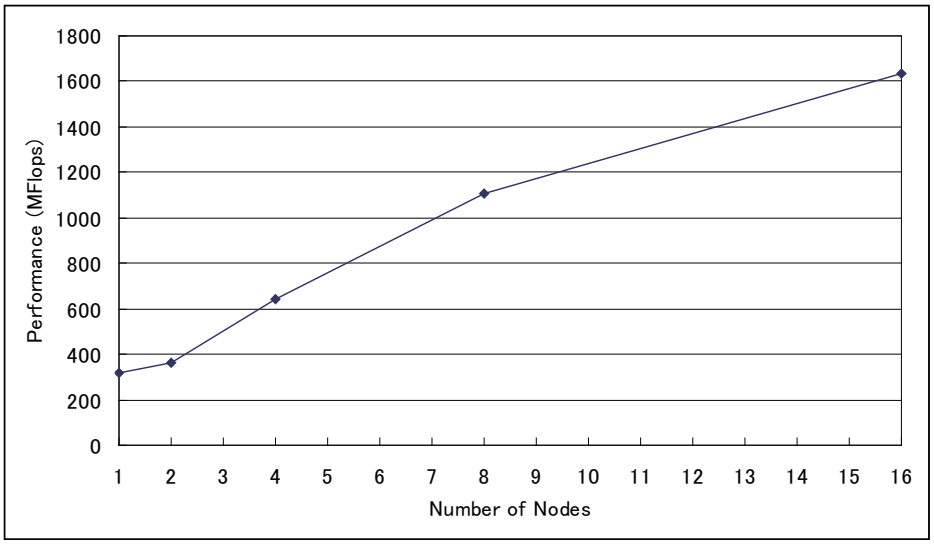

Figure 4. Linpack Performance

E. FFT

The six-step algorithm is used as in MPI version of FT. The one-dimensional vector is accessed as two-dimensional array. In the two-dimensional array, 1D FFT is performed in each direction. During the 1DFFT on each direction, the matrix transpose operations are required three times. The matrix transpose operations are implemented by all-to-all communication of gmove directive and local copy operations. For 1D FFT, FFTE routine (FFT235) is used.

1) Source Code and LOC: We show parallel code of FFT below. The LOC is 217.

- hpccfft.h
  The LOC is 23.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>

#ifdef LONG_IS_64BITS
typedef unsigned long u64Int_t;
typedef long s64Int_t;
#else
typedef unsigned long long u64Int_t;
typedef long long s64Int_t;
#endif

typedef double fftw_real;

typedef struct {
     fftw_real re, im;
} fftw_complex;

#define c_re(c)  ((c).re)
#define c_im(c)  ((c).im)

#define ARR2D(a,i,j,lda) a[(i)+(j)*(lda)]
#define ARR3D(a,i,j,k,lda1,lda2) a[(i)+(lda1)*((j)+(k)*(lda2))]
#define ARR4D(a,i,j,k,l,lda1,lda2,lda3) a[(i)+(lda1)*((j)+(lda2)*((k)+(lda3)*(l)))]
#define c_mul3v(v,v1,v2) \
{ c_re(v) = c_re(v1)*c_re(v2) - c_im(v1)*c_im(v2); c_im(v) = c_re(v1)* c_im(v2) + c_im(v1)
#define c_assgn(d,s) { c_re(d)=c_re(s);c_im(d)=c_im(s); }
```

- fft_xmp.h
  The LOC is 16.

```
#pragma xmp nodes p(*)

#define N1 (10*1024)
#define N2 (10*1024)
#define N (N1*N2)
#define N_MAX (N1 > N2 ? N1 : N2)

#pragma xmp template t0(0:((10*1024)*(10*1024))-1)
#pragma xmp template t1(0:(10*1024)-1)
#pragma xmp template t2(0:(10*1024)-1)

#pragma xmp distribute t0(block) onto p
#pragma xmp distribute t1(block) onto p
#pragma xmp distribute t2(block) onto p
```

```
15
16 extern fftw_complex in[N], out[N], tbl_ww[N];
17 extern fftw_complex tbl_w1[N1], tbl_w2[N2], work[N_MAX];
18
19 #pragma xmp align in[i] with t0(i)
20 #pragma xmp align out[i] with t0(i)
21 #pragma xmp align tbl_ww[i] with t0(i)
```

- zfft1d-sixstep-xmp.c
  The LOC is 104.

```
1 #include "hpccfft.h"
2 #include "fft_xmp.h"
3
4 fftw_complex a_work[N2][N1];
5 #pragma xmp align a_work[*][i] with t1(i)
6
7 void settbl2(fftw_complex w[N1][N2])
8 {
9   int i,j;
10  double pi, px;
11 #pragma xmp align w[i][*] with t1(i)
12
13  pi=4.0*atan(1.0);
14  px=-2.0*pi/((double)(N1*N2));
15
16 #pragma xmp loop on t1(i)
17  for(i = 0; i < N1; i++) {
18    for(j = 0; j < N2; j++) {
19      c_re(w[i][j]) = cos(px*((double)(i))*((double)(j)));
20      c_im(w[i][j]) = sin(px*((double)(i))*((double)(j)));
21    }
22  }
23 }
24
25 void zfft1d0(fftw_complex a[N2][N1], fftw_complex b[N1][N2],
26              fftw_complex ww[N1][N2], fftw_complex w1[N1], fftw_complex w2[N2],
27              fftw_complex *work, int ip1[3], int ip2[3])
28 {
29   int i, j;
30   fftw_complex ztmp1,ztmp2,ztmp3;
31 #pragma xmp align a[i][*] with t2(i)
32 #pragma xmp align b[i][*] with t1(i)
33 #pragma xmp align ww[i][*] with t1(i)
34
35 #pragma xmp gmove
36   a_work[:][:] = a[:][:];
37
38 #pragma xmp loop on t1(i)
39   for(i = 0; i < N1; i++) {
40     for(j = 0; j < N2; j++) {
41       c_assgn(b[i][j],a_work[j][i]);
42     }
43   }
44
```

```
45 #pragma xmp loop on t1(i)
46   for(i = 0; i < N1; i++)
47     HPCC_fft235(b[i],work,w2,N2,ip2);
48
49 #pragma xmp loop on t1(i)
50   for(i = 0; i < N1; i++){
51     for(j = 0; j < N2; j++) {
52       c_assgn(ztmp1,b[i][j]);
53       c_assgn(ztmp2,ww[i][j]);
54       c_mul3v(ztmp3, ztmp1, ztmp2);
55       c_assgn(b[i][j],ztmp3);
56     }
57   }
58
59 #pragma xmp loop on t1(i)
60   for(i = 0; i < N1; i++) {
61     for(j = 0; j < N2; j++){
62       c_assgn(a_work[j][i],b[i][j]);
63     }
64   }
65
66 #pragma xmp gmove
67   a[:][:] = a_work[:][:];
68
69 #pragma xmp loop on t2(j)
70   for(j = 0; j < N2; j++) {
71     HPCC_fft235(a[j],work,w1,N1,ip1);
72   }
73
74 #pragma xmp gmove
75   a_work[:][:] = a[:][:];
76
77 #pragma xmp loop on t1(i)
78   for(i =0; i < N1; i++){
79     for(j=0; j < N2; j++){
80       c_assgn(b[i][j],a_work[j][i]);
81     }
82   }
83 }
84
85 int
86 zfft1d(fftw_complex a[N], fftw_complex b[N], int iopt, int n1, int n2)
87 {
88   int i;
89   double dn;
90   int ip1[3], ip2[3];
91 #pragma xmp align a[i] with t0(i)
92 #pragma xmp align b[i] with t0(i)
93
94   if (0 == iopt) {
95     HPCC_settbl(tbl_w1, n1);
96     HPCC_settbl(tbl_w2, n2);
97     settbl2((fftw_complex **)tbl_ww);
98     return 0;
```

```
99    }
100
101   HPCC_factor235( n1, ip1 );
102   HPCC_factor235( n2, ip2 );
103
104   if (1 == iopt){
105 #pragma xmp loop on t0(i)
106     for (i = 0; i < N; ++i) {
107       c_im( a[i] ) = -c_im( a[i] );
108     }
109   }
110
111   zfft1d0((fftw_complex **)a, (fftw_complex **)b, (fftw_complex **)tbl_ww,
112           (fftw_complex *)tbl_w1, (fftw_complex *)tbl_w2, (fftw_complex *)work,
113           ip1, ip2);
114
115   if (1 == iopt) {
116     dn = 1.0 / N;
117 #pragma xmp loop on t0(i)
118     for (i = 0; i < N; ++i) {
119       c_re( b[i] ) *= dn;
120       c_im( b[i] ) *= -dn;
121     }
122   }
123
124   return 0;
125 }
```

- fft_main_xmp.c
  The LOC is 74.

```
1 #include "hpccfft.h"
2 #include "fft_xmp.h"
3
4 fftw_complex in[N], out[N], tbl_ww[N];
5 fftw_complex tbl_w1[N1], tbl_w2[N2], work[N_MAX];
6
7 double HPL_timer_cputime( void );
8
9 int
10 main(void)
11 {
12   int rv, n, failure = 1;
13   double Gflops = -1.0;
14   FILE *outFile;
15   int doIO = 0;
16   double maxErr, tmp1, tmp2, tmp3, t0, t1, t2, t3;
17   int i, n1, n2;
18   int rank;
19
20 #ifdef _XMP
21   rank = xmp_get_node_num();
22 #else
23   rank = 0;
24 #endif
```

```
25
26    doIO = (rank == 0);
27
28    outFile = stdout;
29    srand(time(NULL));
30
31    n = N;
32    n1 = N1;
33    n2 = N2;
34
35    t0 = -HPL_timer_cputime();
36    HPCC_bcnrand( 0, in );
37    t0 += HPL_timer_cputime();
38
39    t1 = -HPL_timer_cputime();
40    zfft1d( in, out, 0, n1, n2 );
41    t1 += HPL_timer_cputime();
42
43    t2 = -HPL_timer_cputime();
44    zfft1d( in, out, -1, n1, n2 );
45    t2 += HPL_timer_cputime();
46
47    t3 = -HPL_timer_cputime();
48    zfft1d( out, in, +1, n1, n2 );
49    t3 += HPL_timer_cputime();
50
51    HPCC_bcnrand( 0, out );
52
53    maxErr = 0.0;
54 #pragma xmp loop on t0(i)
55    for (i = 0; i < N; i++) {
56      tmp1 = c_re( in[i] ) - c_re( out[i] );
57      tmp2 = c_im( in[i] ) - c_im( out[i] );
58      tmp3 = sqrt( tmp1*tmp1 + tmp2*tmp2 );
59      maxErr = maxErr >= tmp3 ? maxErr : tmp3;
60    }
61
62 #pragma xmp reduction(max:maxErr)
63
64    if (doIO) {
65      fprintf( outFile, "Vector size: %d\n", n );
66      fprintf( outFile, "Generation time: %9.3f\n", t0 );
67      fprintf( outFile, "Tuning: %9.3f\n", t1 );
68      fprintf( outFile, "Computing: %9.3f\n", t2 );
69      fprintf( outFile, "Inverse FFT: %9.3f\n", t3 );
70      fprintf( outFile, "max(|x-x0|): %9.3e\n", maxErr );
71    }
72
73    if (t2 > 0.0) Gflops = 1e-9 * (5.0 * n * log(n) / log(2.0)) / t2;
74
75    if (doIO)
76      fprintf(outFile, "Single FFT Gflop/s %.6f\n", Gflops);
77
78    return 0;
```

```
79 }
80
81 #include <sys/time.h>
82 #include <sys/resource.h>
83
84 #ifdef HPL_STDC_HEADERS
85 double HPL_timer_cputime( void )
86 #else
87 double HPL_timer_cputime()
88 #endif
89 {
90    struct rusage                ruse;
91
92    (void) getrusage( RUSAGE_SELF, &ruse );
93    return( (double)( ruse.ru_utime.tv_sec  ) +
94            ( (double)( ruse.ru_utime.tv_usec ) / 1000000.0 ) );
95 }
```

2) Performance Archieved: We shows performance evaluation results in Figure 5. The vector size is 104857600 ((10x1024) x (10x1024)). The main reason of low performance is inefficient implement of gmove communication function.

node(s)	GFlops
1	0.68225
2	0.74600
4	0.93646
8	1.03960
16	1.27068

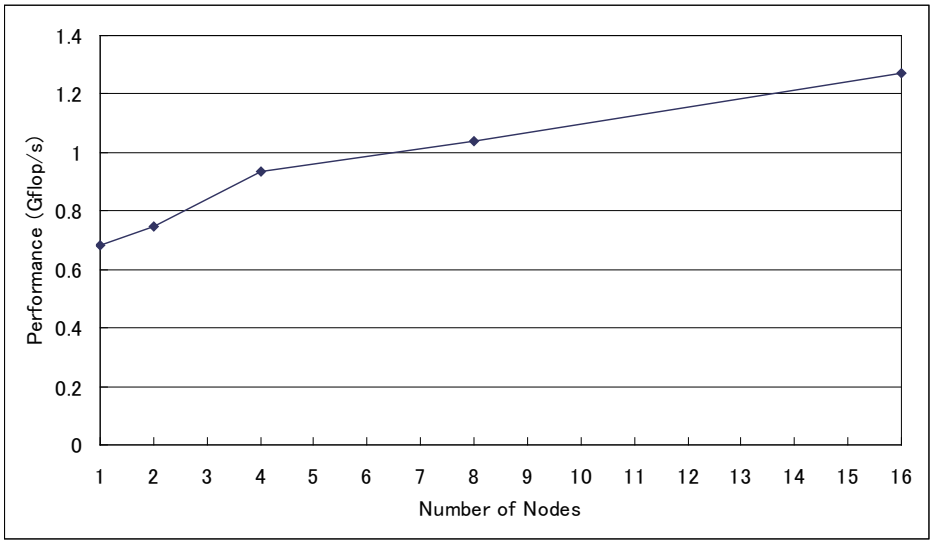

Figure 5. FFT Performance

REFERENCES

[1] XcalableMP, http://www.xcalablemp.org/

[2] CLOC, http://cloc.sourceforge.net/

[3] Coarray Fortran, http://www.co-array.org/