# XcalableMP for Productivity and Performance in HPC Challenge Award Competition Class 2

Masahiro Nakao[1,a)] Hitoshi Murai[2] Takenori Shimosaka[2] Mitsuhisa Sato[1,2]

1. Center for Computational Sciences, University of Tsukuba, Japan
2. RIKEN Advanced Institute for Computational Science, Japan
a) mnakao@ccs.tsukuba.ac.jp

## 1. Summary

In this paper, we present our XcalableMP implementation of the HPCC HPL, RandomAccess, FFT, and the Himeno benchmark [1] which is a typical stencil application.

The highlights of this submission are as follows:

- We implemented three HPCC benchmarks; HPL, RandomAccess, and FFT. In addition, we implemented the Himeno benchmark.
- The SLOC (Source lines of code) of the benchmarks is shown in **Table 1**.
- The performance summary of the benchmarks is shown in **Table 2**.
- To measure performances, we used the K computer at RIKEN AICS and HA-PACS at University of Tsukuba in Japan (**Table 3**).

**Table 1** Source lines of code for HPCC and Himeno benchmarks

| | HPL | Random Access | FFT | Himeno |
|---|---|---|---|---|
| XcalableMP | 288 | 253(278)† | 87(+1,522)‡ | 115 |
| Reference | 8,800 | 938 | 128(+1,904)* | 612** |

† The number in no-brackets is SLOC of the K computer specific version RandomAccess, the number in brackets is one for HA-PACS.
‡ SLOC of the FFT main kernel written in XcalableMP is 87, and SLOC of the FFT interface and another kernels written in C is 1,522.
* We sum up SLOC of all source code in hpcc-1.4.2/FFT/. The number of 128 is SLOC of corresponds to the FFT main kernel of XcalableMP.
** Refer to http://accc.riken.jp/2466.htm#itemid4562

**Table 2** Performance summary of HPCC and Himeno benchmarks achieved on the K computer and HA-PACS

| Code | Machine | Max nodes (× cores) | Performance |
|---|---|---|---|
| HPL | K Computer | 4,096 node (× 8) | 156.5 TFlops(30% of peak) |
| | HA-PACS | 64 node (× 16) | 16.8 TFlops(79% of peak) |
| Random Access | K Computer | 8,192 node (× 8) | 104.3 GUP/s |
| | HA-PACS | 64 node (× 16) | 2.4 GUP/s |
| FFT | K Computer | 1,024 node (× 8) | 986.8 GFlops |
| | HA-PACS | 64 node (× 16) | 219.3 GFlops |
| Himeno | K Computer | 1,024 node (× 8) | 5.8 TFlops |
| | HA-PACS | 64 node (× 16) | 1.6 TFlops |

**Table 3** Specification of each node and software on the experimental environment

| | K computer | HA-PACS* |
|---|---|---|
| CPU | SPARC64 VIIIfx 2.0GHz (Single Socket), 8Cores/Socket | Xeon E5-2670 2.6GHz (Dual Socket), 8Cores/Socket |
| Memory | DDR3 SDRAM 16GB, 64GB/s/Socket | DDR3 SDRAM 128GB, 51.4GB/s/Socket |
| Network | Torus fusion six-dimensional mesh/torus network, 5GB/s | Infiniband QDRx2rails Fat-tree network, 4GB/s |
| C / Fortran Compier | Fujitsu C/Fortran Compiler Version K-1.2.0-09 | Intel Compiler 12.1 |
| BLAS | Fujitsu SSLII Version K-1.2.0-09 | Intel MKL 10.3 |
| Communication Library | Fujitsu MPI Version K-1.2.0-09 | Intel MPI 4.0.3, GASNet 1.18.2 |
| XcalableMP | Omni XcalableMP Compiler 0.6.0-alpha** | |

* HA-PACS has NVIDIA Tesla M2090 as an accelerator device, but we do not use it in this time.
** The official version will be available until SC12 at http://www.xcalablemp.org/.

## 2. Overview of XcalableMP

XcalableMP [2–4], XMP for short, is a directive-based language extension for distributed memory systems, which is proposed by the XMP Specification Working Group consists of members from academia, research laboratories, and industries. It allows users to easily develop parallel programs and to tune performance with minimal and simple notation. A part of the design is based on the experiences of High Performance Fortran (HPF) [5, 6] and Coarray Fortran (CAF) [7].

The features of XMP are as follows:

- XMP supports typical parallelization under "global-view model" programming, and enables parallelizing the original sequential code using minimal modification with simple directives.
- XMP also includes a CAF-like PGAS feature as "local-view model" programming.

- XMP is defined as an extension for familiar languages, such as C and Fortran, to reduce code-rewriting and educational costs.
- The important design principle of XMP is “performance awareness”. All actions of communication and synchronization are taken by directives or coarray syntax, different from HPF.

We have been developing an Omni XMP Compiler as a prototype compiler. The Omni XMP Compiler can compile a XMP C source code and a XMP Fortran source code. Each source code and language is called XMP/C or XMP/Fortran in this paper. The Omni XMP Compiler and a XMP specification are available at a XMP official website (http://www.xcalablemp.org).

# 3. Implementation and Performance of benchmarks

## 3.1 Experimental Settings

This section provides a brief overview of the XMP implementation and performance result of HPL, RandomAccess, FFT, and Himeno benchmark. All benchmarks were compiled using Omni XcalableMP 0.6.0-alpha developed at University of Tsukuba and RIKEN AICS. To evaluate the performance of these benchmarks, we used maximum 65,536 CPU cores of the K computer and maximum 1,024 CPU cores of HA-PACS cluster (Table 3). All source line counts exclude comments and blank lines, but include validation operation and printing performance result (Table 1).

## 3.2 HPL

### 3.2.1 Implementation

The HPL measures the floating point rate of execution for solving a dense system of linear equations. We implemented XMP version HPL written in XMP/C. The points of our implementation are as follows:

□ Block-cyclic distribution

XMP uses a “template” as a virtual array to specify the data distribution. The follow code and **Fig. 1** show that array *A[][]* is distributed on each node in block-cyclic manner. In this code, a **template** directive declares a two-dimensional template *t*, and a **node** directive declares a two-dimensional node set *p*. A **distribute** directive distributes the template *t* onto $P \times Q$ nodes in the same number of blocks. Finally, an **align** directive declares the array *A[][]* and aligns it with the template *t*. Note that a macro does not be used in XMP directive now. In the follow source code, macro *P*, *Q*, *N*, and *NB* are used to explain XMP directives.

```
1 double A[N][N];
2 #pragma xmp nodes p(P,Q)
3 #pragma xmp template t(0:N−1, 0:N−1)
4 #pragma xmp distribute t(cyclic(NB), cyclic(NB)) onto p
5 #pragma xmp align A[i][j] with t(j,i)
```

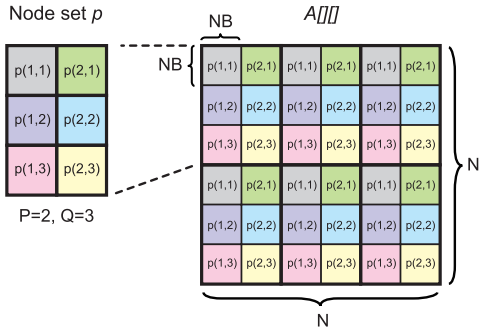


**Fig. 1** Block cyclic distribution

□ Panel broadcast by using **gmove** directive

XMP can access global data using the **gmove** directive while keeping the global image. The XMP/C is extended to support the array section notation to access global data easily. The follow code and **Fig. 2** indicate a Panel broadcast operation. The number before colon in brackets means start elements accessed and the number after colon means the length of elements accessed. Target element block of the array *A[][]* which is blue in Fig. 2 is broadcasted to the array *A_L[][]* which exists on each node. The array *A_L[][]* is also distributed in block-cyclic manner, but only one dimension of the array *A_L[][]* is distributed. A programmer does not need to pack/unpack data for translation because XMP runtime can do it automatically.

```
1 double A_L[N][NB];
2 #pragma xmp align A_L[i][∗] with t(∗,i)
3   :
4 #pragma xmp gmove
5   A_L[j+NB:N−j−NB][0:NB] = A[j+NB:N−j−NB][j:NB];
```

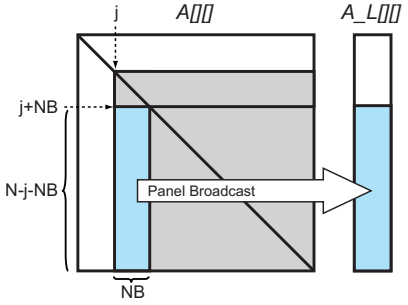


**Fig. 2** Panel broadcast

- Usage of BLAS library for distributed array
  High performance mathematical libraries, such as BLAS, ScaLAPACK, and so on, are often used in computational science. In XMP, a programmer can use these libraries for distributed array defined by XMP directives. XMP has a rule that a pointer of distributed array indicates the local pointer on the node which has its distributed data. The follow code shows that *DGEMM* function applies the distributed array *A[][]*, *A_L[][]*, and *A_U[][]*. The *ltog_y* and *ltog_x* are macros to calculate a global index from a local index.

```
1  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N/Q−local_y, N/P−local_x, NB, −1.0,
2                &A_L[ltog_y(local_y)][0], NB, &A_U[0][ltog_x(local_x)], N/P, 1.0, &A[ltog_y(local_y)][ltog_x(local_x)], N/P);
```

### 3.2.2 Performance

To implement the HPL, we used the BLAS library which is parallelized with *pthread* automatically. Therefore, we executed our implementation with 1 processes 8 threads on one CPU on each machine. The data size used in this evaluation is about 90% of the system memory. For comparison, we also evaluated the hpcc-1.4 HPL on HA-PACS. The performance results are shown in **Table 4** and **Table 5**. On HA-PACS, the performance of the XMP implementation is good and is equal to that of the hpcc-1.4 HPL. However, the performance on the K computer is not good on only one node. Now, we are investigating the cause of the inefficient point.

**Table 4** The performance of HPL on the K computer

| #Cores | Performance (TFlops) | of peak |
|---|---|---|
| 8 | 0.079 | 61.8% |
| 32 | 0.300 | 58.6% |
| 128 | 1.145 | 55.9% |
| 512 | 4.333 | 52.8% |
| 2,048 | 15.736 | 48.0% |
| 8,192 | 52.933 | 40.3% |
| 32,768 | 156.497 | 29.8% |

**Table 5** The performance of HPL on HA-PACS

| #Cores | Performance (TFlops) | | of peak | |
|---|---|---|---|---|
| | XMP | hpcc-1.4 | XMP | hpcc-1.4 |
| 8 | 0.150 | 0.149 | 90.1% | 89.5% |
| 16 | 0.288 | 0.295 | 86.4% | 88.7% |
| 32 | 0.571 | 0.576 | 85.8% | 86.5% |
| 64 | 1,140 | 1.155 | 85.6% | 86.8% |
| 128 | 2.215 | 2.259 | 83.1% | 84.9% |
| 256 | 4.405 | 4.574 | 82.7% | 85.9% |
| 512 | 8.784 | 8.878 | 82.5% | 83.4% |
| 1,024 | 16.795 | 17.680 | 78.9% | 83.0% |

## 3.3 RandomAccess

### 3.3.1 Implementation

RandomAccess benchmark measures the performance of random integer updates of memory. The measurement is Giga UPdates per Second(GUPS). Our algorithm is iterated over sets of CHUNK updates on each node. In each iteration, it calculates for each update the destination node that owns the array element to be updated and communicates the data with each other. This communication pattern is known as the complete exchange or all-to-all personalized communication, which can be performed efficiently by an algorithm called recursive exchange algorithm when the number of nodes is power-of-two [8].

We implemented the algorithm with a set of remote writes to a coarray in local-view programming by using XMP/C. Note that the number of the remote writes is also sent as the additional first element of the data. A point-to-point synchronization is specified with the XMP's **post** and **wait** directives to realize asynchronous behavior of the algorithm.

The following code is a part of our RandomAcesss implementation. Line 2 means that elements from *0* to *nsend* of array *send[]* are put to those from *0* to *nsend* of array *recv[]* in node *partner+1*. In line 3 and 7, the **sync memory** directive ensures the remote definition of a coarray is compete.

```
1   send[isend][0] = nsend;
2   recv[j][0:nsend+1]:[ipartner+1] = send[isend][0:nsend+1]; // Coarray operation
3  #pragma xmp sync_memory
4  #pragma xmp post(p(ipartner+1), 0)
5     :
6  #pragma xmp wait(p(jpartner+1))
7  #pragma xmp sync_memory
```

The coarray feature of XMP on the K computer is based on the extend RDMA interface of its MPI. Since variables on both-hand sides of a coarray operation should be coarrays due to a restriction of the interface, the array *send* on the right-hand side, which was originally a normal (non-coarray) data, is declared as a coarray. On HA-PACS, the restriction does not exist because XMP coarray feature is implemented by GASNet, *send* is declared as a normal array.

### 3.3.2 Performance

On the K computer and HA-PACS, we performed our implementation of RamdomAccess on each CPU core, as it is called flat-MPI. The performance result is shown in **Table 6** and **Table 7**. For comparison, we also evaluated the custom RandomAccess whose functions *sort_data()* and *update_table()* were optimized by Fujitsu, which was used HPC Challenge class 1 in 2011. The string "Fujitsu" in Table 6 and Table 7 indicates this RandomAccess. We implemented two XMP RandomAccess codes based on this RandomAccess. The first one was optimized for the K computer, the latter one was the same algorithm of the Fujitsu version RandomAccess. The latter one is used to evaluate its performance on HA-PACS. SLOC and algorithm of both implementations are a bit different (Table 1). The data size used in

this evaluation is equal to 1/4 of the system memory.

Table 6 and Table 7 show that the performance of Fujitsu RandomAccess is a little better than that of XMP implementation.

**Table 6** The performance of RandomAccess on the K computer

| #Cores | Performance (GUP/s) | |
|---|---|---|
| | XMP | Fujitsu |
| 8 | 0.08 | 0.08 |
| 64 | 0.70 | 0.62 |
| 512 | 2.08 | 2.41 |
| 4,096 | 11.41 | 13.55 |
| 32,768 | 61.43 | 75.08 |
| 65,536 | 104.31 | 120.24 |

**Table 7** The performance of RandomAccess on HA-PACS

| #Cores | Performance (GUP/s) | |
|---|---|---|
| | XMP | Fujitsu |
| 1 | 0.07 | 0.07 |
| 4 | 0.15 | 0.16 |
| 16 | 0.28 | 0.32 |
| 64 | 0.47 | 0.70 |
| 128 | 0.93 | 1.52 |
| 1,024 | 2.38 | 3.55 |

## 3.4 FFT

### 3.4.1 Implementation

FFT benchmark measures the floating point rate of execution for double precision complex 1-dimensional Discrete Fourier Transform. We parallelized a "pzfft1d.f" in ffte-5.0 [10] by using XMP/Fortran in global-view model. The follow source code is a part of in "xmp-pzfft1d.f90" which is a XMP version "pzfft1d.f".

```
1  !$XMP nodes p(*)
2  !$XMP template tx(NX)
3  !$XMP template ty(NY)
4  !$XMP distribute tx(block) onto p
5  !$XMP distribute ty(block) onto p
6  !$XMP align A(*,i) with ty(i)
7  !$XMP align A_WORK(i,*) with tx(i)
8  !$XMP align B(*,i) with tx(i)
9      :
10 !$XMP gmove
11   A_WORK(1:NX:1,1:NY:1) = A(1:NX:1,1:NY:1) ! all-to-all
12
13 !$XMP loop (i) on tx(i)
14   DO 70 I=1,NX
15     DO 60 J=1,NY
16       B(J,I)=A_WORK(I,J)
17     60 CONTINUE
18   70 CONTINUE
```

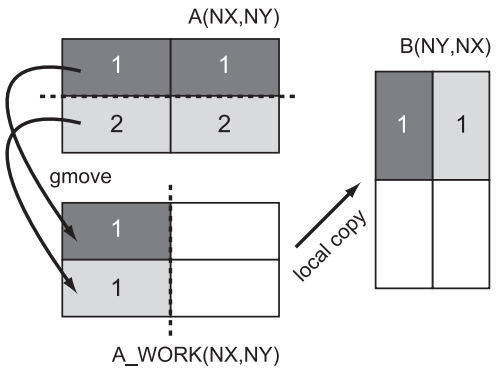


**Fig. 3** Matrix transpose in FFT

In line 2 to 8, the **template**, **distribute**, and **align** directives are describing the distribution of arrays in a block manner. In six-step FFT, matrix transpose operation is done before 1-dimensional 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 (line 10 to 11). **Fig. 3** 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** directive is written to collect those elements. A new array *A_WORK()* is declared to store the elements. *A_WORK()* is distributed by template *tx* 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 in line 13 to 18. The rest of the code is simple work-mapping parallelizing loop statements by using XMP **loop** directive.

The SLOC of "xmp-pzfft1d.f90" is only 87 (the SLOC of original "pzfft1d.f" is 158). Note that, our implementation of FFT uses C interface and another kernels of the original FFT in hpcc-1.4 and ffte-5.0. In fact, the function *pzfft1d()* in "pzfft1d.c" calls the function *xmp_pzfft1d0()* in "xmp-pzfft1d.f90". Hence, total SLOC is the almost the same as original one, but the kernel function in "xmp-pzfft1d.f90" is simple. Furthermore, this implementation is a good demonstration how to mix a XMP program with a MPI program.

### 3.4.2 Performance

On the K computer, we performed our implementation with 2 processes 8 threads on one node. On HA-PACS, we performed it with flat-MPI. The performance is shown in **Table 8** and **Table 9**. For comparison, we also evaluated the hpcc-1.4 FFT. Note that Omni XMP compiler only supports integer size of template now. Hence, an array which has a large number of elements cannot be distributed [*1]. In this evaluation, we used the vector 16,777,216(512MB) length per process on the K computer and HA-PACS.

Table 8 and Table 9 show that the performance of XMP implementation is near to that of hpcc 1.4 FFT written in MPI. The main reason of this difference is that the main loops in *pzfft1d()* is parallelized by OpenMP in MPI implementation, and not in XMP implementation.

[*1] This limitation will be removed sometime in the near future.

Although XMP has an option for hybrid parallelism, the current compiler has some problems for it. In the final presentation, we will be able to report the results by hybrid parallelism. Another reason is that our implementation does not block data when transposing. It is not difficult to implement this blocking mechanism for XMP, but the source code will become similar to the hpcc-1.4 FFT.

**Table 8** The performance of FFT on the K computer

| #Cores | Performance (GFlops) | | of peak | |
|---|---|---|---|---|
| | XMP | hpcc-1.4 | XMP | hpcc-1.4 |
| 8 | 0.91 | 1.58 | 0.71% | 1.23% |
| 32 | 3.47 | 6.08 | 0.68% | 1.19% |
| 128 | 13.84 | 27.11 | 0.67% | 1.32% |
| 512 | 45.76 | 112.68 | 0.56% | 1.37% |
| 2,048 | 191.26 | 334.86 | 0.58% | 1.02% |
| 8,192 | 986.81 | 1,312.17 | 0.75% | 1.00% |

**Table 9** The performance of FFT on HA-PACS

| #Cores | Performance (GFlops) | | of peak | |
|---|---|---|---|---|
| | XMP | hpcc-1.4 | XMP | hpcc-1.4 |
| 1 | 0.71 | 0.67 | 3.43% | 3.24% |
| 4 | 2.68 | 2.78 | 3.22% | 3.34% |
| 16 | 7.69 | 8.71 | 2.31% | 2.61% |
| 64 | 18.91 | 24.02 | 1.42% | 1.80% |
| 256 | 74.87 | 80.93 | 1.40% | 1.52% |
| 1,024 | 219.33 | 238.39 | 1.03% | 1.11% |

## 3.5 Himeno benchmark

### 3.5.1 Implementation

The Himeno benchmark evaluates performance of incompressible fluid analysis code using the Jacobi iteration method. The reason of selecting this benchmark is a good example of a stencil application benchmark and to demonstrate parallelization by XMP **shadow** and **reflect** directives which communicate and synchronize the overlapped regions.

A part of the Himeno benchmark written in XMP/Fortran is shown in the follow source code. In line 5, a **shadow** directive declares a shadow region of the distributed array *p*. The **shadow** directive specifies the width of the shadow region. In line 7, the **reflect** directive synchronizes data of the shadow region onto the neighboring nodes before referring array *p* by loop iteration. This parallelization is very simple and straightforward. Basically, a programmer only has to add XMP directives into the sequential version Himeno benchmark.

```
1  !$xmp nodes n(2,2)
2  !$xmp template t(mimax,mjmax,mkmax)
3  !$xmp distribute t(*,block,block) onto n
4  !$xmp align (*,j,k) with t(*,j,k) :: p, bnd, wrk1, wrk2
5  !$xmp shadow p(0,1,1)
6      :
7  !$xmp reflect (p)
8  !$xmp loop (J,K) on t(*,J,K) reduction (+:GOSA)
9    do K = 2, kmax−1
10     do J = 2, jmax−1
11       do I = 2, imax−1
12         S0 = a(I,J,K,1)*p(I+1,J,K) + ...
13         SS = (S0*a(I,J,K,4)−p(I,J,K))*bnd(I,J,K)
14         GOSA = GOSA + SS * SS
15           :
16       enddo
17     enddo
18   enddo
```

This benchmark measures performance to proceed major loops in solving the Poisson's equation solution in Flops. To verify the result of this benchmark, it calculates the residual of the Jacobi iteration method.

### 3.5.2 Performance

On the K computer, we used an automatic thread-parallelization to evaluate it performance (8 threads per process). On HA-PACS, we measured the performance with flat-MPI. We set the number of array *p* elements of each node(weak scaling) is $256 \times 64 \times 64$ on the K computer, the number of array *p* elements of each node is $512 \times 64 \times 64$ on HA-PACS. This size is able to obtain good performance because of a cache effect.

The performance result is shown in **Table 10** and **Table 11** . For comparison, we also evaluated the original Himeno benchmark [1] which is written in MPI Fortran.

**Table 10** The performance of Himeno on the K computer

| #Cores | Performance (GFlops) | | of peak | |
|---|---|---|---|---|
| | XMP | Original | XMP | Original |
| 8 | 12.43 | 18.38 | 9.71% | 14.36% |
| 32 | 39.78 | 67.52 | 7.77% | 13.19% |
| 128 | 127.81 | 231.07 | 6.24% | 11.28% |
| 512 | 342.82 | 929.41 | 4.18% | 11.35% |
| 2,048 | 1,495.55 | 3,713.52 | 4.56% | 11.33% |
| 8,192 | 5,773.18 | 14,688.12 | 4.40% | 11.21% |

**Table 11** The performance of Himeno on HA-PACS

| #Cores | Performance (GFlops) | | of peak | |
|---|---|---|---|---|
| | XMP | Original | XMP | Original |
| 1 | 5.20 | 5.83 | 24.98% | 28.01% |
| 4 | 19.32 | 24.05 | 23.23% | 28.91% |
| 16 | 33.92 | 34.97 | 10.19% | 10.51% |
| 64 | 129.13 | 136.72 | 9.70% | 10.27% |
| 256 | 513.46 | 534.24 | 9.64% | 10.03% |
| 1,024 | 1,571.91 | 1,571.66 | 7.38% | 7.38% |

The result of the HA-PACS in Table 11 indicates that the performance of XMP is equal to that of original. However, Table 10 shows that the performance of XMP is worse than that of original on the K computer. Now, we are investigating why the performance is worse on the K computer.

## 4. Conclusion

This report has investigated the productivity and the performance of the XMP PGAS language through the HPCC and Himeno benchmarks. XMP has a rich set of functions based on global-view and local-view model that allows it to develop applications with a smaller cost. On the K computer, the performance of the HPL and Himeno benchmarks written in XMP are not satisfactory. We are still working on improving the performance by removing these programs, and we hope to have a chance to present our results on BoF at SC12.

**Acknowledgments** The specification of XMP has been designed by the XMP Specification Working Group of PC Cluster consortium, Japan. This study was supported by the "Seamless and Highly productive Parallel Programming Environment for High performance computing" project funded by the Ministry of Education, Culture, Sports, Science and Technology, Japan. and is supported by the G8 Research Councils Initiative.

### References

[1] The Riken Himeno CFD Benchmark: `http://accc.riken.jp/2444.htm`
[2] The XcalableMP Website: `http://www.xcalablemp.org`
[3] Masahiro Nakao, Jinpil Lee, Taisuke Boku, and Mitsuhisa Sato: Productivity and Performance of Global-View Programming with XcalableMP PGAS Language, CCGrid 2012 - The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, Ottawa, Canada, May, 2012.
[4] Jinpil Lee: A Study on Productive and Reliable Programming Environment for Distributed Memory System, March, 2012.
[5] C.H. Koelbel, D.B. Loverman, R. Shreiber, GL. Steele Jr., M.E. Zosel. The High Performance Fortran Handbook, MIT Press, 1994.
[6] Ken Kennedy, Charles Koelbel,Hans Zima: The rise and fall of High Performance Fortran: an historical object lesson, Proceedings of the third ACM SIGPLAN conference on History of programming languages, Pages 7-1-7-22, 2007
[7] R. Numwich and J. Reid: Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.
[8] R. Ponnusamy, A. Choudhary and G. Fox: Communication Overhead on CM5: An Experimental Performance Evaluation, Proc. Frontiers '92, pp.108–115, 1992.
[9] HPC Challenge Website: `http://icl.cs.utk.edu/hpcc/software/index.html`
[10] FFTE: A Fast Fourier Transform Package: `http://www.ffte.jp`