International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:10, No:3, 2016

# Implementation of ADETRAN Language Using Message Passing Interface

Akiyoshi Wakatani

*Abstract*—This paper describes the Message Passing Interface (MPI) implementation of ADETRAN language, and its evaluation on SX-ACE supercomputers. ADETRAN language includes pdo statement that specifies the data distribution and parallel computations and pass statement that specifies the redistribution of arrays. Two methods for implementation of pass statement are discussed and the performance evaluation using Splitting-Up CG method is presented. The effectiveness of the parallelization is evaluated and the advantage of one dimensional distribution is empirically confirmed by using the results of experiments.

*Keywords*—Iterative methods, array redistribution, translator, distributed memory.

## I. Introduction

MPI is usually utilized as a defact standard for programming on distributed memory multicomputers such as PC clusters and supercomputers, while PGAS (Partitioned Global Address Space) language is getting popular in order to improve both of a productivity of programs and an effective performance. PGAS language, which includes Coarray Fortran [1], X10 [2], Chapel [3] and XcableMP [4], has the characteristic that arrays are distributed over a multicomputer and the distributed arrays can be transparently accessed from every node of the multicomputer. Namely, the effective performance can be improved by reducing the number of data transfers between nodes, and the productivity of program can be improved by accessing distributed arrays transparently and then simplifying programs without MPI type communication like Send/Recv. For example, Coarray FORTRAN is implemented by using MPI to achieve a high performance [5]. On the other hand, the authors developed ADETRAN language based on FORTRAN language for ADENART machine and evaluated its performance [6]. ADENART is a distributed memory multicomputer equipped with a network that efficiently redistributes two-dimensional arrays and three-dimensional arrays at the same cost as memory accesses. ADETRAN language is a FORTRAN based language that includes a statement for data distribution based on data parallel, a statement for data redistribution and a mechanism of overlapping computation with communication. Although ADETRAN language is different from PGAS languages, it is one of approaches that exploit data locality and improve program productivity.

This paper describes the implementation policy of ADETRAN language by using MPI and presents the performance evaluation. The availability of ADETRAN language on distributed memory multicomputers is discussed and the translator of ADETRAN language is also mentioned.

A. Wakatani is with the Faculty of Intelligence and Informatics, Konan University, Kobe, 6588501, Japan (e-mail: wakatani@konan-u.ac.jp; http://pplinux.is.konan-u.ac.jp).

## II. ADETRAN Language

### A. ADENART

ADI (Alternating Direction Implicit) is an iterative method for a partial differential equation that tries to converge a solution with changing the direction to solve. It is known that this method is easy to parallelize because other dimensions except for one dimension are solved independently, but array redistribution is needed on distributed memory multicomputers. For example, three-dimensional array element `A(i,j,k)` is the `i`-th data on `(j,k)` processor on the x-direction, and then it should be redistributed to the `j`-th data on `(k,i)` processor on the y-direction if necessary. ADENART is a distributed memory multicomputer equipped with a network that is suitable for such a redistribution. In addition, since such a network is also suitable for Splitting Up CG (Conjugate Gradient) method [7] as well as ADI, ADENART is used for fluid dynamic simulations and plasma simulations. Therefore, ADETRAN language is designed so that array redistribution is easily described.

### B. Pdo Statement

Hereafter the case of three-dimensional arrays is described. In ADETRAN language, an array element with "/" specifies a distributed array. In Fig. 1, array element `A(i,/j,k/)` is the `i`-th data on `(j,k)` processor.

In the figure, pdo statement specifies parallel executions. Namely, on Ny × Nz processors, loop iterations (`i=1, Nx`) are executed with array elements distributed in the same way and scalar variables.

```
double precision A(Nx,Ny,Nz)
pdo j=1,Ny,k=1,Nz
   do i=1,Nx
    A(i,/j,k/)=A(i,/j,k/)*2.0
   end do
pend
```

Fig. 1 Pdo Statement

### C. Pass Statement

The network of ADENART is designed so that array redistribution is easily carried out. Pass statement specifies the array redistribution.

In Fig. 2, pass statement specifies that the `i`-th data on `(j,k)` processor is redistributed as the `j`-th data on `(k,i)` processor. So, after the pass statement, array element `A(i,j,k)` is stored as both of `A(i,/j,k/)` and `A(i/,j,/k)`.

```
pass i=1,Nx,j=1,Ny,k=1,Nz
  A(i/,j,/k)=A(i,/j,k/)
pend
```

Fig. 2 Pass Statement

### D. S-Scheme

Although ADENART is equipped with a network suitable for array redistribution, the communication overhead of the redistribution may cause the degradation of effective performance. In order to amortise the overhead, ADENART has the overlapping mechanism that a signal is emitted when a substitution to the data is completed and then a send operation starts as soon as the signal is found. This mechanism is called S-scheme, which theoretically hides the communication time behind the calculation time. To utilize the S-scheme, ADETRAN compiler should find the consecutive pair of pdo statement and pass statement and should produce the instructions for the S-scheme. Fig. 3 shows the example where S-scheme can be carried out.

```
pdo j=1,Ny,k=1,Nz
   do i=1,Nx
    A(i,/j,k/)=A(i,/j,k/)*2.0
   enddo
pend
pass i=1,Nx,j=1,Ny,k=1,Nz
  A(i/,j,/k)=A(i,/j,k/)
pend
```

Fig. 3 S-Scheme

Namely, just after the substitution of `A(1,/j,k/)` is completed, the data transfer of `A(1,/j,k/)` to `A(1/,j,/k)` is started, which is overlapped with the computation of `A(2,/j,k/)`. If there is enough computation, the overhead of the communication can be completely hidden.

## III. Implementation Policy

Our implementation policy is so simple that the translator can be easily created for any distributed memory multicomputers. We do not exploit any specific feature of the multicomputers. Note that, in this paper, C language based implementation is described, but our approach can be easily extended to FORTRAN language based implementation.

### A. Implementation of Pdo Statement

The distribution of an array element is determined by the location of "/". When three-dimensional array `A(Nx,Ny,Nz)` is distributed over $P(=P1 \times P1)$ processors, A_x[Nz/P1][Ny/P1][Nx], A_y[Nx/P1][Nz/P1][Ny] and A_z[Ny/P1][Nx/P1][Nz] are allocated on each processor. So, `A(i,/j,k/)` is defined as `A_x[k%P1][j%P1][i]` on processor `(j/P1, k/P1)`, and `A(i/,j,/k)` is defined as `A_y[i%P1][k%P1][j]` on processor `(k/P1, i/P1)`.

According to the above array representation, an x-directional pdo statement shown in Fig. 1 is roughly implemented as follows:

```
for(int _k=0; _k<Nz/P1; _k++){
  for(int _j=0; _j<Ny/P1; _j++){
   for(i=0; i<Nx; i++){
    A[_k][_j][i]=A[_k][_j][i]*2.0;
   }
  }
}
```

Fig. 4 Implementation of pdo statement

The reason why variables `j` and `k` are expressed as `_j` and `_k` in Fig. 4 is that variables `j` and `k` must be actually expressed as $j = id1*(Ny/P1) + \_j$ and $k = id2*(Nz/P1) + \_k$.

### B. Implementation of Pass Statement

In the case of the redistribution between x-direction and y-direction shown in Fig. 2, one-to-one communication from `(j,k)` processor to `(k,i)` processor should be carried out. In practice, each processor sends data of $(Ny/P1) \times (Nz/P1) \times (Nx/P1)$ as a group. Since each processor sends data to P1 processors and receives data from P1 processors for the array redistribution, the number of communication activations must be totally $P1 \times P1 \times P1$. On the other hand, `(j,k)` processor sends data to `(k,i)` processor and then the destination of P1 processors (`(j,k)` `j`=0...P1) is the same, so we have an alternative method, That is, these P1 processors first send data to `(k,k)` processor by using gather communication that is one of MPI collective communications, and then `(k,k)` processor distributes data to P1 processors (`(k,i)` `i`=0...P1) by using scatter communication that is also one of MPI collective communications. Therefore, the number of communication activations can be reduced to $2 \times P1 \times P1$ compared with the first method. However, although the latter method can reduce the number of communication activations, the latter method requires two collective communications and thus total data size of the array redistribution is large. In addition, since only one communication can be hidden behind the computation by

$$
\begin{aligned}
\boldsymbol{h}^n &= C^{-1}(A\boldsymbol{u}^n - \boldsymbol{f}) \\
\tau &= \frac{(\boldsymbol{h}^n, C\boldsymbol{h}^n)}{(\boldsymbol{d}^n, A\boldsymbol{d}^n)} \\
\boldsymbol{u}^{n+1} &= \boldsymbol{u}^n + \tau\boldsymbol{d}^n \\
\boldsymbol{h}^{n+1} &= \boldsymbol{h}^n + \tau C^{-1}A\boldsymbol{d}^n \\
\beta &= \frac{(\boldsymbol{h}^{n+1}, C\boldsymbol{h}^{n+1})}{(\boldsymbol{h}^n, C\boldsymbol{h}^n)} \\
\boldsymbol{d}^{n+1} &= -\boldsymbol{h}^{n+1} + \beta\boldsymbol{d}^n
\end{aligned}
$$

(a) Iterative method

$$
\begin{aligned}
C &= (D+X)D^{-1}(D+Y)D^{-1}(D+Z) \\
D^{-1} &= [(\mathbf{0},\mathbf{0},\mathbf{0})(\mathbf{0},(0,1/6,0),\mathbf{0})(\mathbf{0},\mathbf{0},\mathbf{0})] \\
D+X &= [(\mathbf{0},\mathbf{0},\mathbf{0})(\mathbf{0},(-1,6,-1),\mathbf{0})(\mathbf{0},\mathbf{0},\mathbf{0})] \\
D+Y &= [(\mathbf{0},\mathbf{0},\mathbf{0})((0,-1,0),(0,6,0),(0,-1,0))(\mathbf{0},\mathbf{0},\mathbf{0})] \\
D+Z &= [(\mathbf{0},(0,-1,0),\mathbf{0})(\mathbf{0},(0,6,0),\mathbf{0})(\mathbf{0},(0,-1,0),\mathbf{0})]
\end{aligned}
$$

(b) Preconditioning

Fig. 5 Splitting up CG method

using S-scheme mentioned later, half of the communications cannot be hidden. Therefore, we take the first method on our implementation.

### C. Implementation of S-Scheme

As mentioned before, since ADENART has the overlapping mechanism, that a signal is emitted when a substitution to the data is completed and then a send operation starts as soon as the signal is found, S-scheme is easily implemented. However, distributed memory multicomputers generally do not have such a mechanism, and it is better to send data on MPI as a group instead of individually because the overhead of communication activations can be amortized to improve the total performance. Thus, since the size of data to be sent by each processor is $(\mathrm{Ny}/\mathrm{P1}) \times (\mathrm{Nz}/\mathrm{P1}) \times (\mathrm{Nx}/\mathrm{P1})$, a non-blocking send communication is started as soon as the substitutions of the data size are completed, and this communication is overlapped with the subsequent substitutions of the data size. Namely, `(j,k)` processor starts a non-blocking communication as soon as the data that are sent to `(k,0)` processor are calculated, and the communication is overlapped with the calculations of data that are sent to `(k,1)` processor. Moreover, after the calculations are completed, the processor starts the non-blocking communication, which is overlapped with the calculations of data that are sent to `(k,2)` processor. Note that `(k,0)` processor receives data from P1 processors at first, and then `(k,1)` processor receives data from P1 processors and so on. So, the completion of the data reception is not simultaneous.

## IV. Evaluation

We evaluate our approach on SX-ACE (NEC) super computers installed at Osaka University. SX-ACE consists of 512 nodes that contains 4 vector CPUs and 64 GB memory (256 GB/s), and the data is transferred between the nodes at the speed of 8 GB/s. The theoretical peak performance of the CPU is 64 GFLOPS, so the total performance reaches 132 TFLOPS [8].

### A. Evaluation Using SPCG Method

Three-dimensional Laplace equation is as follows: $A\boldsymbol{u} = \boldsymbol{f}$ $(Au_{ijk} \equiv 6u_{ijk} - u_{i-1jk} - u_{i+1jk} - u_{ij-1k} - u_{ij+1k} - u_{ijk-1} - u_{ijk+1})$. For solving the Laplace equation, we utilize a preconditioned conjugate gradient iterative method, called SPCG (Splitting Up Conjugate Gradient). We translate ADETRAN program of SPCG into FORTRAN program with MPI extensions by hand, and the FORTRAN program is used to evaluate our approach on SX-ACE [7]. Fig. 5 shows SPCG method and the preconditioning for SPCG method. Here, $\boldsymbol{u}^n$ is unknown array at the $n$-th iteration, $\tau_n$ and $\beta_n$ are scalar parameters for the iterative method, $\boldsymbol{h}^n$ is a residual vector and $\boldsymbol{d}^n$ is a direction vector that is used for updating the unknown array.

As shown in Fig. 5, the preconditioning is suitable for parallel computation because the preconditioning can be solved independently in each direction, but array redistributions are required repeatedly between different directions in distributed memory multicomputers.

The elapsed time of 100 iterations executed on the system of 4 nodes is measured for different array sizes with varying the number of MPI processes and OpenMP threads. Table I shows the results. It should be noted that the product of the number of MPI processes and the number of OpenMP threads per process must be up to 16, and MPI communication is not used when the number of MPI processes is 1.

When the array size is small $(48 \times 48 \times 48)$, the case of 4 MPI processes and 4 OpenMP threads per process is faster than the case of 16 MPI processes. Otherwise, the minimal elapsed time is achieved with 16 MPI processes. However, in all cases, the minimal elapsed time is achieved with 1 MPI process that does not include data transfer between nodes, so the effectiveness of parallelization using MPI processes is not confirmed. The reason is that the current algorithm and implementation repeatedly utilizes array redistributions, and the cost of the communications more strongly alleviates the effectiveness of parallelization than expected. We will discuss the improved version of the implementation to avoid such a difficulty later. In addition, the S-scheme is applied to the results, but the performance gain is very limited. We will discuss the effectiveness of the S-scheme later.

TABLE I
ELAPSED TIME OF SPCG METHOD (SEC.)

| node | MPI process | OpenMP thread | $48 \times 48 \times 48$ | $192 \times 192 \times 192$ | $384 \times 384 \times 384$ |
|---|---|---|---|---|---|
| 4 | 16 | 1 | 0.210 | 4.45 | 39.25 |
| | 4 | 4 | 0.170 | 5.61 | 112.49 |
| | 4 | 1 | 0.210 | 7.37 | 313.74 |
| 1 | 4 | 1 | 0.166 | 7.34 | 348.89 |
| | 1 | 4 | 0.045 | 3.04 | 28.96 |
| | 1 | 1 | 0.094 | 7.15 | 93.12 |

### B. Evaluation of S-Scheme

As mentioned before, the effectiveness of the S-scheme is limited. The reason is that the computation part that hides the communication overhead is the preconditioning and the product of matrix and vector ($A\boldsymbol{u}$) in the case of SPCG method, and thus the computation part is too small to hide all the communication overhead of array redistributions. So, the effectiveness of the S-scheme is small. Then, in order to discuss the effectiveness of the S-scheme when the computational complexity is large, we consider 6 cases which include pdo statement with different computational complexity. The computational complexity of cal1 is the smallest (2 add operations), and the computational complexity of cal6 is the largest (60 function calls). Elapsed times in the case of the array size of $192 \times 192 \times 192$ are measured.

As shown in table, the effectiveness of the S-scheme is confirmed when the computational complexity is large. For example, when one MPI process resides on a node and 3 OpenMP threads are executed on one process, the elapsed time in the case of cal6 is reduced to 20.6 seconds from 22.56 seconds. The reason is that since each node consists of 4 CPU cores, 3 cores are in charge of computation and one core is in charge of operations for the non-blocking communication, so the elapsed time can be reduced. This effectiveness is large in the case of a large computational complexity, but the S-scheme degrades the performance in the case of cal1 and cal2, because the cost of overhead of the S-scheme exceeds the reduction of the elapsed time caused by the S-scheme. On the other hand, when 4 MPI processes are carried out on each node, the effectiveness of the S-scheme is very slight. For example, the elapsed times in the case of cal6 are 21.21 seconds and 21.15 seconds, and the elapsed times in the case of cal5 are 11.70 seconds and 11.74 seconds. The reason is that each core is very busy for calculating and thus any core is not in charge of receiving operation. If a core is receiving data of the non-blocking communication, it cannot carry out computation concurrently. Therefore, we have two alternatives; one core must be in charge of communication in order to keep computation concurrent with communication; all cores must be in charge of computation and the S-scheme is not adopted. As mentioned later, our implementation of the translator does not utilize the S-scheme.

### C. One-Dimensional Distribution

As mentioned before, the performance of the simple parallelization does not overcome that of single node (4 threads), unfortunately. In order to avoid such a difficulty, three-dimensional arrays are distributed not over a two-dimensional processor array, but over a one-dimensional processor array. Namely, as shown in Fig. 6, arrays are distributed in z-direction, and each processor keeps two-dimensional distributed arrays in local. The advantage of this configuration is that the number of array redistributions can be reduced to 3 from 2, and thus the elapsed time can decrease. It should be noted that when the size of the corresponding dimension is too small, the effective parallelism may be limited, but a large size simulation problem can provide enough parallelism. Table III shows the results of experiments using this configuration.

In both the cases of the array size of $192 \times 192 \times 192$ and $384 \times 384 \times 384$, the elapsed time on 8 nodes (32 CPUs) using one-dimensional distribution is smaller than the elapsed time on 1 node (1 thread), especially the speedup of over 6 is achieved in the latter case. Although the achieved speedup is not enough from a point of view of the number of CPUs, the reduction of the number of array redistributions contributes to the performance enhancement very much. In the future, in order to alleviate the communication overhead, the algorithm should be reconsidered and the effectiveness of our approach should be enhanced.

```
double precision A(Nx,Ny,Nz)
pdo k=1,Nz
  do j=1,Nz
   do i=1,Nx
    A(i,j,/k/)=A(i,j,/k/)*2.0
   enddo
  enddo
pend
```

Fig. 6 Pdo statement with 1 dimensional distribution

### D. Translator

According to the implementation policy described above, we develop ADETRAN translator that deals with one-dimensional distribution and produces a program without S-shceme feature. Fig. 7 shows the snapshot of the translator.

This translator is a prototype, but ifall statement that calculates a conjunction of distributed LOGICAL arrays and ifany statement that calculates a disjunction of distributed LOGICAL arrays are available, so some practical application programs can be processed by the translator. The translator

TABLE II
EFFECTIVENESS OF S-SCHEME (SEC.)

| | 4 nodes × 4 MPIs × 1 thread | | 4 nodes × 1 MPI × 3 threads | | 1 node × 4 MPIs × 1 thread | |
|---|---|---|---|---|---|---|
| | no S-scheme | w/S-scheme | no S-scheme | w/S-scheme | no S-scheme | w/S-scheme |
| cal1 | 2.09 | 2.14 | 3.30 | 3.54 | 4.68 | 4.76 |
| cal2 | 2.90 | 2.97 | 3.88 | 4.33 | 7.99 | 7.88 |
| cal3 | 5.29 | 5.34 | 6.44 | 5.89 | 17.38 | 14.81 |
| cal4 | 7.94 | 7.92 | 8.80 | 9.19 | 28.60 | 24.20 |
| cal5 | 11.74 | 11.70 | 13.66 | 12.27 | 46.33 | 37.38 |
| cal6 | 21.21 | 21.15 | 22.56 | 20.60 | 90.63 | 71.13 |

TABLE III
ELAPSED TIME OF SPCG METHOD WITH 1 DIMENSIONAL DISTRIBUTION (SEC.)

| node | MPI | OpenMP | 192 × 192 × 192 | 384 × 384 × 384 |
|---|---|---|---|---|
| 4 | 16 | 1 | 4.45 | 39.25 |
| 4 (1 dim.) | 16 | 1 | 2.85 | 22.93 |
| 8 (1 dim.) | 32 | 1 | 2.19 | 15.82 |
| 1 | 1 | 4 | 3.04 | 28.96 |
| 1 | 1 | 1 | 7.15 | 93.12 |

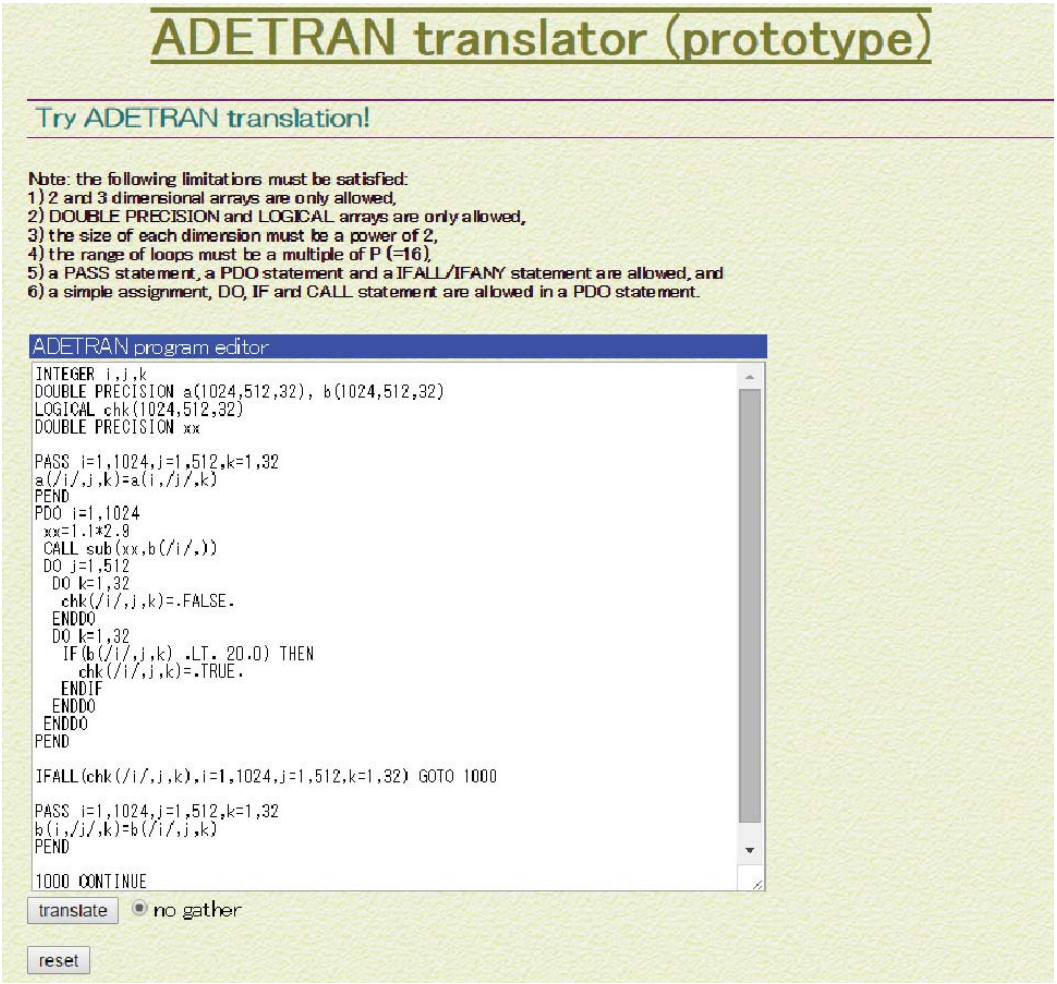


Fig. 7 ADETRAN translator

is constructed by using bison and flex, and produces a C program with MPI extension through lexical analysis, syntax analysis and semantic analysis [9]. The default number of MPI processes is 16, but it can be easily changed to an arbitrary

number by rewriting a define macro statement [10].

## V. Conclusion

This paper proposes MPI implementation policy of ADETRAN language, and presents the results of the evaluation on SX-ACE supercomputers. ADETRAN language easily expresses array distributions, so the translation method of such an expression into MPI expression is described and two methods for array redistributions are discussed. According to our performance evaluation, the effectiveness of parallelization for SPCG iterative method can be enhanced by using one-dimensional distribution that is improved version of our implementation.

In the future, we will improve the optimization of communications for enhancing the effectiveness of parallelization. In addition, we will apply our approach to other languages besides FORTRAN and will develop a new language suitable for the SPCG type algorithm.

## Acknowledgments

We are grateful to Professor Tatsuo Nogi of Kyoto University for helpful discussions. We received generous support from Professor Shinji Odanaka of Osaka University for using SX-ACE supercomputers. The author would like to express his gratitude to both professors.

Part of this research was supported by JSPS KAKENHI Grant Number 15K00501 (2015-2017). This research was also supported in part by MEXT, Japan.

## References

[1] R. Numrich and J. Reid, "Co-array Fortran for parallel programming," *ACM SIGPLAN Fortran Forum*, vol. 17, issue 2, pp. 1-31, 1998.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Praun and V. Sarkar "X10: an object-oriented approach to non-uniform cluster computing," *ACM SIGPLAN Notices*, vol. 40, issue 10, pp. 519-538, 2005.

[3] B. Chamberlain, D. Callahan and H. Zima, "Parallel Programmability and the Chapel Language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291-312, 2007.

[4] M. Nakao, H. Murai, T. Shimosaka and M. Sato, "Productivity and Performance of the HPC Challenge Benchmarks with the XcalableMP PGAS Language", in *Proc. of the 7th International Conference on PGAS Programming Models*, pp. 157-171, 2013.

[5] C. Yang, W. Bland, J. Mellor-Crummey, P. Balaji, "Portable, MPI-interoperable coarray fortran," *ACM SIGPLAN Notices*, vol. 49, issue 8, pp. 81-92, 2014.

[6] H. Kadota, K. Kaneko, I. Okabayashi, T. Okamoto, T. Mimura, Y. Nakakura, A. Wakatani, M. Nakajima, J. Nishikawa, K. Zaiki and T. Nogi "Pallel computer ADENART - its architecture and application," in *Proc. of the 5th international conference on Supercomputing*, pp. 1-8, 1991.

[7] S. Odanaka and T. Nogi, "Massively parallel computation using a splitting-up operator method for three-dimensional device simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, issue 7, pp. 824-832, 1995.

[8] "SX-ACE super computer," http://www.hpc.cmc.osaka-u.ac.jp/en/sx-ace/ (as of 3/2/2016)

[9] "GNU bison," http://www.gnu.org/software/bison/ (as of 3/2/2016)

[10] "ADETRAN translator," http://pplinux.is.konan-u.ac.jp/atran.html (as of 3/2/2016)

**Akiyoshi Wakatani** Akiyoshi Wakatani was born in Osaka City, Osaka Pref., Japan, on February 3, 1962. He received the B. Eng. degree from the Department of Applied Mathematics and Physics, Faculty of Engineering, Kyoto University, Kyoto, Japan, in 1984. He received the M. Eng. degree from the Division of Applied Systems Science, Faculty of Engineering, Kyoto University in 1986. He also received the Dr. Eng. degree from the Division of Information Engineering, Faculty of Engineering, Kyoto University in 1996. He was with Matsushita Electric Industrial (currently Panasonic) from 1986 to 2000, as a researcher and a senior researcher. From 1992 to 1994, he was a visiting scholar of Oregon Graduate Institute of Science and Technology, OR, USA. From 2000 to 2006, he was an Associate Professor of Department of Information Science and Systems Engineering, Faculty of Science and Engineering, Konan University, Kobe Japan. From 2006 to 2008, he was an Full Professor of the same department in the same university. Since 2008, he has been a Full Professor of Department of Intelligence and Informatics, Faculty of Intelligence and Informatics, Konan University, Kobe Japan. His research interest includes parallel processing and programming education.