2. Programming Environment Research Team

2.1. Team members

Mitsuhisa Sato (Team Leader)
Hitoshi Murai (Research Scientist)
Tetsuya Abe (Postdoctoral Researcher)
Swann Perarnau (Visiting Researcher, JSPS Research Fellow)
Tomotake Nakamura (Research Associate)
Takenori Shimosaka (Research Associate)
Masahiro Yasugi (Visiting Researcher)
Tomio Kamada (Visiting Researcher)
Hitoshi Sakagami (Visiting Researcher)
Hiroaki Umeda (Visiting Researcher)
Miwako Tsuji (Visiting Researcher)
Horitz Helias (Visiting Researcher)
Susanne Kunkel (Visiting Researcher)
Masahiro Nakao (Visiting Researcher)
Tomoko Nakashima (Assistant (Concurrent))

2.2. Research Activities

The K computer system running in AICS is a massively parallel system which has a huge number of processors connected by the high-speed network. In order to exploit full potential computing power to carry out advanced computational science, efficient parallel programming is required to coordinate these processors to perform scientific computing. We conducts researches and developments on parallel programming models and language to exploit full potentials of large-scale parallelism in the K computer and increase productivity of parallel programming.

In 2012FY, in order to archive these objectives above, we carried out the following researches:

1) We continued the development of XcalableMP(XMP) programming languages. XcalableMP is a directive-based language extension which allows users to develop parallel programs for distributed memory systems easily and to tune the performance by having minimal and simple notations. The specification has been designed by XcalableMP Specification Working Group (XMP Spec WG) which consists of members from academia and research labs to industries in Japan. We have been working with XMP Spec WG to improve the specification. In this year, we released XMP Fortran, and deployed it to the K computer. According to the results from a preliminary performance evaluation using a XMP version of the SCALEp code (a climate code for LES), we improved the runtime system on the K computer. We conducted the evaluation of

XMP programs on the K computer using HPCC benchmark and submitted the results to SC12 HPCC class 2 competition, and were selected as a finalist. We also continued the design of the interface to MPI programs in XMP, and IO supports of the XMP language.

2) For the research for performance tuning tools for large-scale scientific applications running on the K computer, we have ported the Scalasca performance turning and analysis tool developed by JSC, Germany, to the K computer. Our update on the Scalasca for the K computer was included in the formal release.
3) We investigated methods and tools to support a correct parallel program. We proposed a light-weight XMP verification tool which helps users to verify XMP programs using descriptions of global-view programming directives in XMP. And also, we studied the model checking technique of the PGAS language including XMP.
4) The processor of the K computer has an interesting hardware mechanism called “sector cache”, which allows partition of L2 on-chip cache to optimize the locality for important data. We published a paper about the technique to optimize the usage of sector cache.
5) We joined Japan-France project FP3C, "Framework and Programming for Post Petascale Computing", from this year. In this year, we worked on porting and performance evaluation of the integrated programming environment of XcalableMP and YML which is developed by the French team.
6) We conducted several collaborations on the performance evaluation with JSC, University of Tsukuba and other groups.

2.3. Research Results and Achievements

2.3.1. Development of XcalableMP and Performance Evaluation on the K computer

We continued the development of XcalableMP compiler in collaboration with University of Tsukuba. At the last SC12, the version 0.6 which includes the implementation of XMP Fortran as well as XMP C has been released. Several members of our team joined the XMP specification Working Group to update the XMP language specification to version 1.1 which was published at SC12.

Several performance evaluation of XcalableMP programs were conducted on the K computer. One of the important results was that of HPC Challenge Benchmarks. We have submitted the results to the SC12 HPC Challenge Benchmark, Class2 Competition, which was selected as a finalist.

In the evaluation of HPCC Benchmarks, we presented our XcalableMP implementation of the HPCC HPL, RandomAccess, FFT, and the Himeno benchmark which is a typical stencil application.

We have measured the performance of HPCC benchmark on two systems: The K computer, upto 8129 nodes, and HA-PACS system at University of Tsukuba, upto 64 nodes. Table 1 shows the

specifications of two systems. Figure 1,2,3,4 shows the performance and scalability of each benchmark respectively.

	The K computer	HA-PACS
CPU	SPARC64 VIIIfx 2.0GHz 8Cores, **128GFlops**	Xeon E5-2670 2.6GHz x2 8Cores x2, **332.8GFlops**
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**

Table 1. Specification of K computer and HA-PACS

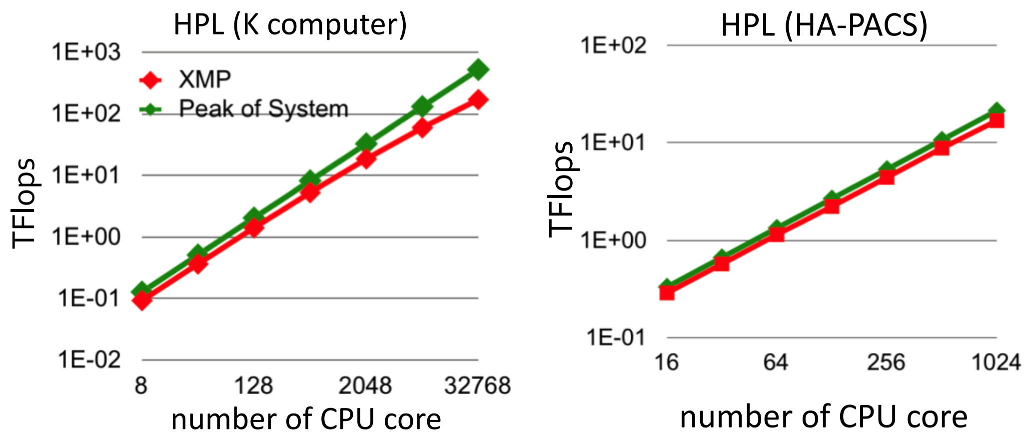

Fig 1. Performance and Scalability of HPL

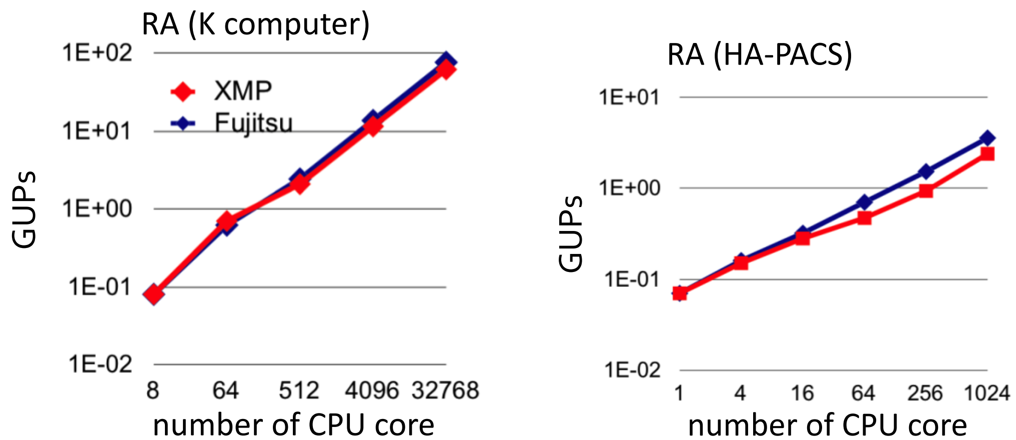

Fig.2 Performance and Scalability of Random Access

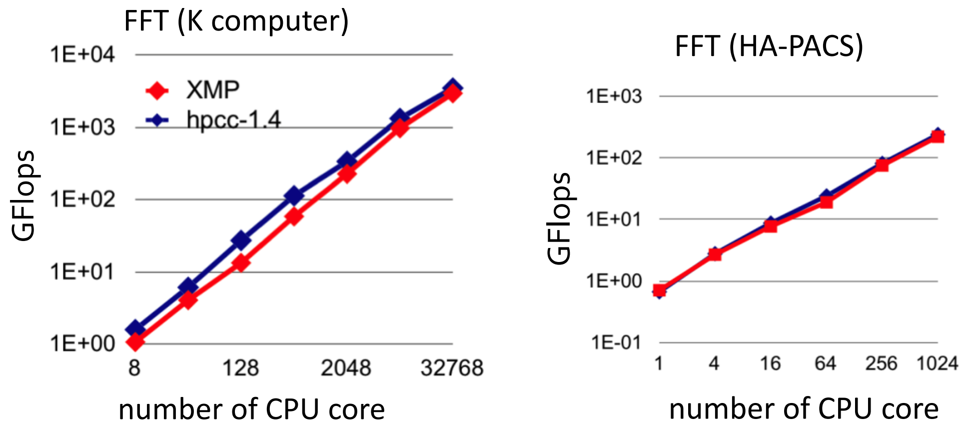

Fig. 3 Performance and Scalability of FFT

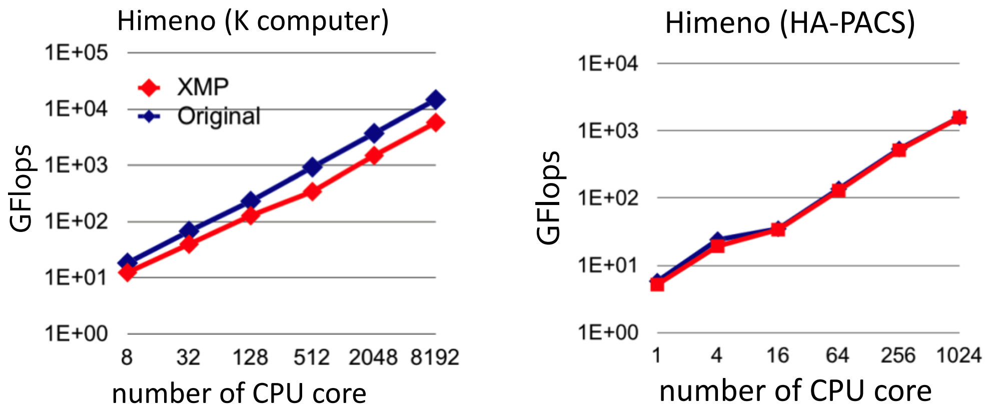

Fig.4 Performance and Scalability of Himeno

Overall, we found the performance gap between the XMP implementation and its counterpart in the K computer, while in HP-PACS the performance of the XMP implementation is very close to the original one. We already found and fixed some problems of the XMP in the run-time system and the algorithms.

Other important factor of programming languages is programability: how easy to program by the language. Figure 5 shows the source line of code (SLOC) of each benchmark programs in XMP comparing to the original MPI implementation as a reference. It means that the XMP can help the programmers to write programs with less lines, resulting in high programmability.

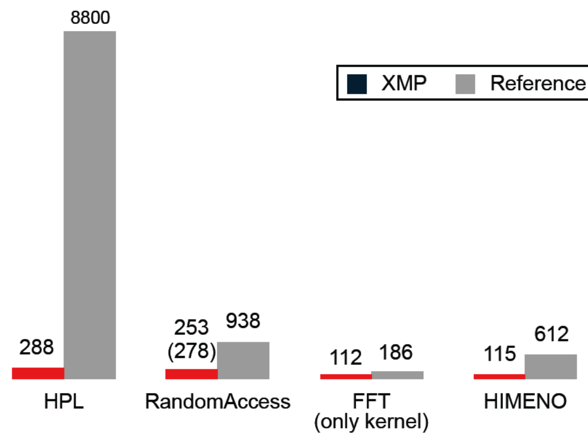

Fig. 5. Source Line of Code (SLOC)

2.3.2. Porting Scalasca performance tool to the K computer

We have ported Scalasca performance tools developed by Julich Supercomputer Center and German Research School for Simulation Sciences, and University of Tennessee, on the K computer. It is designed to analyze parallel application execution behavior on large-scale systems with many thousands of processors such as the K computer. It offers an incremental performance-analysis procedure that integrates runtime summaries with in-depth studies of concurrent behavior via event tracing, adopting a strategy of successively refined measurement configurations.

Our effort for porting the Scalasca to the K computer was included in the latest release 1.4.3. We have tested the tool to analyze the performance of CG in NAS Parallel benchmark using 16,384 nodes on the K computer. The view of the tool is shown in Figure 6.

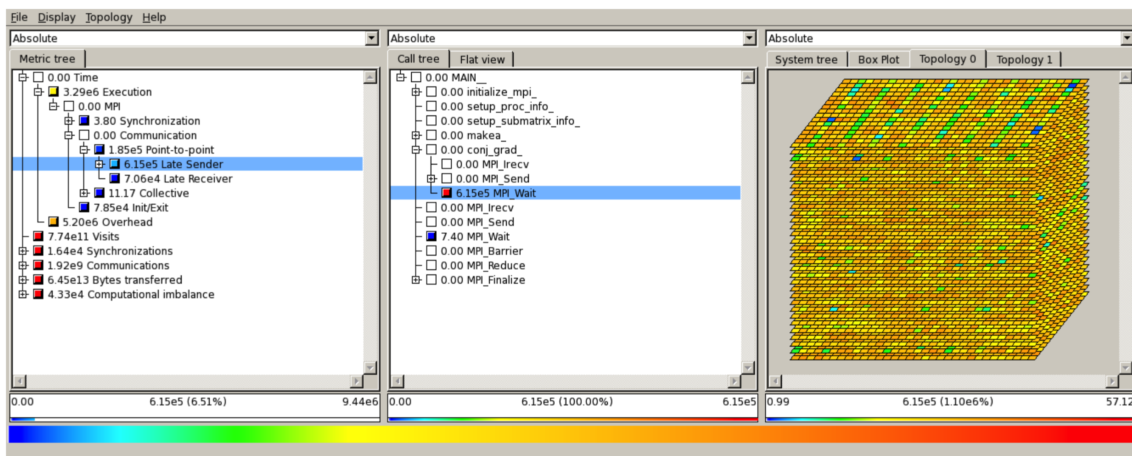

Fig. 6. View of Scalasca performance tool on K computer

(NPB CG benchmark, 32x32x16, 16,384 nodes)

2.3.3 Program Verification Techniques for PGAS programming models

In XcalableMP (XMP), programmers can include explicit synchronizations by adding directives to their source code. In this sense, XMP provides programmers with performance awareness. As such, part of the performance of programs can be attributed to the programmers, i.e., XMP requires interactive programming by the programmers.

We developed a tool that alerts programmers to missing and redundant synchronizations they have included referring, respectively, to non-local array indices and a decrease in the performance of programs. The tool uses XMP directives, making programs more structured, and on-the-fly checks whether directives are missing or redundant, while programmers are editing their programs.

Consider the code fragment shown below, the first pragma line specifies the broadcast operation of data a from node p(1) to nodes from p(2) to p(10), and the second pragma lines performs the broadcast the data from node p(2) to nodes from p(11) to p(20). Then, the third line which performs the broadcast from p(1) to p(10) and p(10), can be detected as a redundant operation. by our tool.

```
#pragma xmp bcast (a) from p(1) on p(2:10)
#pragma xmp bcast (a) from p(2) on p(11:20)
#pragma xmp bcast (a) from p(1) on p(10:11)
```

In the second example shown below, the array a is declared as a distributed array with block distribution. In the XMP execution, each node executes the same code independently if no xmp pragmas are specified. This means the assignment `b=a[0]` is executed on every nodes, and it causes the error on nodes where `a[0]` is not allocated. This kind of errors can be easily detected by static semantics check of our tool.

```
#pragma xmp distribute t(block) onto p
int i;
int a[100];
#pragma xmp align a[i] with t(i)
...
b=a[0];
```

We designed a light-weight static checking program for our tool, which is implemented by using a parser combinator library Parsec, and user-defined datatypes in Haskell performing pattern-matchings by constructors of the user-defined data types. And the tool is integrated with GNU Emacs editor. The pragma and the line containing passible errors are high-lighted in the editor, as shown in Figure 7.

```
emacs@ABET-VAIO
File Edit Options Buffers Tools XcalableMP C Help
#pragma xmp nodes p(2)
#pragma xmp template t(9)
#pragma xmp distribute t(block) onto p
int i;
int b[9];
#pragma xmp align b[i] with t(i)
#pragma xmp shadow b[1:2]

int a[9];

int main(void){

#pragma xmp reflect (b) width [1:2]
  a[0]=b[0];
#pragma xmp reflect (b) width [1:3]
#pragma xmp reflect (b) width [1:2]

#pragma xmp bcast (a) from p(1) on p(2)
#pragma xmp bcast (a) from p(2) on p(3)
#pragma xmp bcast (a) from p(1) on p(3)

#pragma xmp lock (a[1]:[1])
#pragma xmp lock (a[1]:[1])
#pragma xmp unlock (a[1]:[1])

  return 0;
}

-(Unix)---  redundant.c    All (28,0)     (XcalableMP/C Dio Abbrev)
MUDA!  Possibly redundant synchronizations.
```

Figure 7. Integrated environment with Emacs

Through the development of our tool, we found that abstract descriptions in XMP are useful to not only development of a program but also verification of the program.

One of the problems with PGAS languages including XcalableMP is that programmers can easily introduce concurrency bugs into their programs. For example, race conditions tend to occur because a portion of a single address space can be manipulated by multiple threads simultaneously. A solution that avoids race conditions is to synchronize accesses from multiple threads with synchronization primitives (e.g., synchronization locks), but this is not that easy for two reasons. First, excessive use of synchronization may severely degrade the performance of the program. Second, synchronization primitives themselves sometimes introduce other problems. For example, improper use of synchronization locks may cause deadlock.

Model checking is one approach for addressing the problem of concurrency bugs. Basically, software model checking explores all the states that can be reached by executing a given program, and checks whether a given property is ensured (e.g., there are no race conditions or deadlock).

Model checking of partitioned global address space programs tends to suffer from the state explosion problem because these programs allow concurrent and/or parallel execution and memory sharing. To avoid this problem, it is essential to perform proper abstractions based on the properties to be verified because these can dramatically reduce the number of states to be explored in programs. However, it is not always easy to automatically infer proper abstractions because programs and properties to be verified vary.

To address the problem, we proposed a model checking framework that includes user-definable

abstractions. The key idea of the framework is that it exposes the intermediate representation of the program's abstract syntax tree, enabling users to define their own abstractions flexibly and concisely by creating a translator for the tree. By allowing users to create translators at the level of abstract syntax trees, the cost of implementing the translator is lower than that of coding text-based pattern matching and rewriting rules because unnecessary redundant rules that often appear in the conventional approach are eliminated.

In addition, we have designed our proof-of-concept implementation of the proposed approach: CAF-SPIN. CAF-SPIN is a software model checking tool for Coarray Fortran (CAF) programs. In the current implementation of CAF-SPIN, both its intermediate representation of abstract syntax trees and translator are written in Haskell. Thus, users are able to define their abstractions by simply writing Haskell functions. Several experimental results using CAF-SPIN were conducted. The experimental results confirmed that abstractions can be defined easily and concisely in CAF-SPIN, and the number of states to be explored for model checking is dramatically reduced with the abstractions.

It is worth noting that, similar to UPC-SPIN, the current CAF-SPIN does not handle relaxed memory models directly, except for sequential consistency. This limitation does not cause problems for checking programs that perform synchronization when accessing shared memory, but may cause problems if synchronizations are accidentally or intentionally omitted for performance reasons. Supporting relaxed memory models is a future work.

2.3.4 "Sector Cache" optimization for the K computer

The processor architecture available on the K computer (SPARC64 VIIIfx) features a hardware cache partitioning mechanism called sector cache. This facility enables software to split the memory cache in two independent sectors: data loads in one sector cannot trigger the eviction of data in the second one. Moreover, software is responsible for data placement in each sector by issuing special instructions tagging the various memory loads performed during execution. The implementation details of this cache partitioning mechanism also enable fast redistribution of the cache during an application's runtime, without any cost, allowing any optimization using the sector cache to be applied multiple times, with different setups, in the event of phase changes.

Unfortunately, in its current state, the compilers provided on the K computer do not implement any automatic optimization using this cache facility. In the contrary, the only high-level interface to this mechanism is a set of directive to instruct the compiler to generate tagging instructions over a code region. Thus, only application programmers with intricate knowledge of both the memory access patterns of their code and the K computer architecture can take advantage of this facility.

To address this issue and to study new optimization schemes using cache partitioning, we investigated a framework using binary instrumentation and reuse distance analysis to discover the

locality of important data structures in an application and to suggest appropriate data distribution schemes for the sector cache. These optimizations are then translated into calls to the source-level API provided by the K computer compilers.

Our framework leverages and extends several existing methodologies. First, we use binary instrumentation of the target application along with debug information parsing to trace the various memory accesses to major data structures of a code region. This trace is then analyzed using a derivative of reuse distance to assess the locality of theses structures. Third, by modeling the impact of these localities on the performance of the application, we identify whether cache thrashing could be reduced by isolating some of these data structures to a specific sector. We envision these components as steps in an optimization loop: after identifying cache performance hotspots, a developer can analyze them, use the sector cache API to optimize them and repeat the process as much as required.

We applied our framework to analyze and optimize a set of HPC benchmarking applications and demonstrate significant performance improvements.

We analyzed two benchmarks from the Omni OpenMP C version 2.3 of the NAS Parallel Benchmarks: CG, LU, and applied our framework. These benchmarks were only using one thread. In both cases, significant optimizations were found.

Most of the computation time of the CG benchmark is spent inside the conj_grad function. This function does not call any other, and is repeated multiple times during the benchmark's lifetime. The core of this function is a sparse matrix-vector product, with most of the memory accesses touching 3 data structures: the sparse matrix a, the column index colidx and a dense vector p. We analyzed the locality of these structure and, unsurprisingly, our framework indicated that the p vector could benefit for isolation using the sector cache. Indeed, both other structures exhibit streaming access patterns due to indirect accesses that could impact negatively the caching of p. Our optimization thus isolates p in sector 1, with enough space to allow good caching. Such optimization, adding only two lines to the source code of the benchmark reduces the execution time of this function by 10%.

Our process to analyze and optimize the LU benchmark was as follows. First, LU spends almost all of its runtime in the ssor function. This function contains a loop, calling successively several subroutines over shared data structures. Iteratively, these calls solves a system of Navier-Stokes equations by successive over relaxation, decomposing it into lower and upper triangular matrices. Overall, eight structures are of interest here: flux, u, rsd and frct, which are global arrays used as input and results storage, and a,b,c and d which are working arrays used across subroutines to hold partial results (triangular matrices). To analyze this benchmark, we configured our framework to trace recursively all instructions of the ssor function or of any other function called from it. The resulting analysis identified each of a, b, c and d to benefit from the sector cache in the same way.

The cache requirements of the other arrays could not fit in any sector configuration. While isolating only one of the 4 arrays identified by our framework only improved by 2% the benchmark's execution time, another optimization gave more interesting results. Indeed, protecting those 4 arrays for streaming accesses to the other variables of the program by pushing them all together in sector one proved to be a better optimization. It resulted in a 8% reduction of execution time of this function. We should note that, as the ssor function passes these arrays to some subroutines as parameters, we had to change the sector cache directives in them to match the actual parameter names. Overall, code modification added 10 lines of directives: for each of the 5 functions, one line for sector size and one for variable isolation. We excluded two functions (rhs and l2norm) from these modifications, as they do not use these arrays.

Table 2 describes the exact optimization on each benchmark's functions, and the resulting improvements. Note that the cache misses reduction reported are direct cache misses: cache misses triggered by the speculative hardware prefetcher are ignored.

Benchmark	Function	Isolated Variables	Sector Size	Miss Reduction (%)	Runtime Reduction (%)
CG	conj_grad	p	(1,11)	19	10
LU	ssor	a,b,c,d	(2,10)	48	8
	blts	ldz,ldy,ldx,d		75	10
	buts	d,udx,udy,udz		18	3
	jacld	a,b,c,d		64	14
	jacu	a,b,c,d		57	6

Table 2.Results of Optimization of NAB benchmarks (GC and LU)

2.4. Schedule and Future Plan

At the end of 2012FY, we have released the XcalableMP C and Fortran, and Scalasca for the users of the K computer. The important goal is to organize collaborations with application developers and improve our software. As one of activities for this goal, we have a plan to organize hands-on meeting with them. Through these case studies, we will extend it for valuable performance analysis in the K computer.

As a research agenda especially for the K computer, we will focus on the design on one-sided communication using K computer's RDMA hardware. We expect that it will contribute the scalability of large-scale applications for the K computer.

2.5. Publication, Presentation and Deliverables

(1) Journal Papers

- None

(2) Conference Papers

1. Masahiro Nakao, Hitoshi Murai Takenori Shimosaka Mitsuhisa Sato. "XcalableMP for Productivity and Performance in HPC Challenge Award Competition Class 2", SC12 The 2012 HPC Challenge Awards BoF, Salt Lake City, Utah, USA, Nov., 2012.
2. Tatsuya Abe, Toshiyuki Maeda, and Mitsuhisa Sato., "Model checking with user-definable abstraction for partitioned global address space languages.", In Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models (PGAS), Online. Santa Barbara, October 2012.
3. Tatsuya Abe and Mitsuhisa Sato. On-the-fly synchronization checking for interactive programming in XcalableMP. In Proceedings of the 5th International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2), pages 29--37. Pittsburgh, September 2012.
4. Tatsuya Abe and Mitsuhisa Sato. "Auto-tuning of numerical programs by block multi-color ordering code generation and job-level parallel execution.", In Proceedings of the 7th International Workshop on Automatic Performance Tuning (iWAPT), volume 7851 of Lecture Notes in Computer Science. Springer, Kobe, July 2012.
5. Swann Perarnau and Mitsuhisa Sato, "Discovering Cache Partitioning Optimizations for the K Computer," in proceedings of APPLC'13 workshop, Shenzhen, 2013.

[not refereed, in Japanese]

1. Tomotake Nakamura and Mitsuhisa Sato, "Performance Analysis using the performance analysis tool Scalasca on the K computer", IPSJ SIG Technical Report (in Japanese), Vol. 2012-HPC-135, pp.1-7, 2012.
2. Hitoshi Murai, Takenori Shimosaka and Yoshiyuki Ohono, Hisashi Yashiro, Hirofumi Tomita and Mitsuhisa Sato, "Performance Evaluation of Parallel Programming Language XcalableMP on the K Computer", IPSJ SIG Technical Report (in Japanese), 2012-HPC-135(44), (2012).
3. Toward Automated Cache Partitioning for the K Computer, Swann Perarnau and Mitsuhisa Sato, IPSJ SIG Technical Report , Okinawa, 2012.

(3) Invited Talks

1. Mitsuhisa Sato, "The next step for Post-Petascale Computing in Japan", HPC in Asia Workshop, ISC 2011, June 2012.
2. Mitsuhisa Sato, "XcalableMP PGAS parallel programming language for productive high-performance scientific programming", International Top-level Forum on Engineering Science and Technology Development Strategy, 2012 International Forum on HPC Challenges in China, Oct 2012

3. Mitsuhisa Sato, "The K computer and XcalableMP parallel language project --- Towards programming environment for productive high-performance scientific programming ---", ComPAR 2013, Jan 2013

(4) Posters and presentations

1. Hitoshi Murai, Masahiro Nakao, Takenori Shimosaka, Mitsuhisa Sato. "Implementation and Evaluation of HPC Challenge Benchmarks with the Omni XcalableMP compiler", The 3rd AICS International Symposium, Kobe, Hyogo, Japan, Feb, 2013.
2. Tatsuya Abe and Mitsuhisa Sato. Auto-tuning of numerical programs by block multi-color ordering code generation and job-level parallel execution. Poster Session in HPC in Asia Workshop, Online. Hamburg, June 2012.

(5) Patents and Deliverables

1. XcalableMP compiler ver. 0.6 (C and Fortran95) for the K computer
2. Scalasca performance analysis tool for the K computer
3. Xcrypt for the K computer
4. GASnet one-sided communication Library for the K computer (test version)