Chapter 4

HPC Usability Research Team

4.1 Members

Toshiyuki Maeda (Team Leader)

Masatomo Hashimoto (Research Scientist)

Itaru Kitayama (Technical Staff I)

Yoshiki Nishikawa (Visiting Scientist, University of Tokyo)

Yves Caniou (Visiting Scientist, Université Claude Bernard Lyon 1)

Judit Gimenez (Visiting Scientist, Barcelona Supercomputing Center)

Sameer Shende (Visiting Scientist, University of Oregon)

Fabien Delalondre (Visiting Scientist, École Polytechnique Fédérale de Lausanne)

Pramod Kumbhar (Visiting Scientist, École Polytechnique Fédérale de Lausanne)

Tatsuya Abe (Visiting Scientist, Chiba Institute of Technology)

Yumeno Kusuhara (Assistant)

4.2 Research Activities

The mission of the HPC Usability research team is to research and develop a framework and its theories/technologies for liberating large-scale HPC (high-performance computing) to end-users and developers. In order to achieve the goal, we conduct research in the following three fields: computing portal systems, virtualization, and program analysis/verification.

4.2.1 Computing Portal

In a conventional HPC usage scenario, users live in a closed world. That is, users have to play roles of software developers, service providers, data suppliers, and end users. Therefore, a very limited number of skilled HPC elites can enjoy the power of HPC, while the general public sometimes gives a suspicious look to the benefit of HPC. In order to address the problem, we are designing and implementing a computing portal framework that lowers the threshold for using, providing, and aggregating computing/data services on HPC systems, and liberates the power of HPC to the public.

4.2.2 Virtualization

Virtualization is a technology for realizing virtual computers on real (physical) computers. One big problem of the abovementioned computer portal that can be used by wide range of users simultaneously is how to ensure safety, security, and fairness among multiple users and computing/data service providers. In order to solve the problem, we plan to utilize the virtualization technology because virtual computers are isolated from each other, thus it is easier to ensure safety and security. Moreover, resource allocation can be more flexible than the conventional job scheduling because resource can be allocated in a find-grained and dynamic way. We also study lightweight virtualization techniques for realizing virtual large-scale HPC for test, debug, and verification of computing/data services.

4.2.3 Program Analysis/Verification

Program analysis/verification is a technology that tries to prove certain properties of programs by analyzing them. By utilizing software verification techniques, we can prove that a program does not contain a certain kind of bug. For example, the byte-code verification of Java VM ensures memory safety of programs. That is, programs that pass the verification never perform illegal memory operations at runtime. Another big problem of the abovementioned computing portal framework is that one computing service can be consists of multiple computing services that are provided by different providers. Therefore, if a bug or malicious attack code is contained in one of the computing services, it may affect the whole computing service (or the entire portal system). In order to address the problem, we plan to research and develop software verification technologies for large-scale parallel programs. In addition, we also plan to research and develop a performance analysis and tuning technology based on source code modification history.

4.3 Research Results and Achievements (FY2012 $\sim$ FY2016)

In this section, we summarize all the research results and achievements from FY2012 to FY2016 because the current research activities of the HPC Usability Research Team have come to an end in FY2016 and the new activities start from FY2017.

4.3.1 Design and Implementation of a Computing Portal Framework for HPC

As a first step of designing and implementing a computing portal framework that can be used by wide range of users, in FY2012, we designed an experimental API/protocol for computing services. More specifically, we designed APIs/protocols that handle registration of services and their providers, registration and authentication of users for each registered service, invocation of computing services, data sharing among multiple computing services, and so on.

The APIs/protocols are designed in such a way to work with the current popular web-based application frameworks (e.g., HTTP, JSON, etc.). Therefore, in theory, we can write programs that utilize multiple computing services in various programming languages (e.g., Ruby, Python, JavaScript, etc.). In addition, a computing service can be registered and published by writing a simple XML file, provided that the application programs of the computing service are installed on the backend system.

In FY2013, based on the prototype design and implementation of a computing portal framework of FY2012, we actually developed a prototype user-interface for the computing portal framework. More specifically, we implemented a web interface that runs on users web browsers and directly communicates with the backend system of the computing portal under the protocol (also designed in FY2012). With the web interface, software developers can easily publish their applications installed in HPC systems. For example, developers can specify the paths to the executables of their applications, parameters of their applications, and so on, via the web interface. In addition, developers can manage user accounts that are allowed to execute their applications. With the same web interface, users are also able to launch jobs. For example, users can select an application published in the computing portal, make an application to developers for using it, launch jobs by executing the application with arbitrary parameters, and manage the launched/exited jobs.

One feature of the implementation is that the communication protocol between the framework and its clients is based on the popular web-based application frameworks (e.g., WebSockets, JSON, etc.). Therefore, developers can develop their own custom interfaces for their applications if the web interface of our framework does not satisfy their requirements. Another distinguishing feature of our framework is that users can use portable

devices (e.g., smartphones, mobile tablets, and so on) because our web interface is carefully designed so that it can be viewed and accessed with any screen size.

In FY2014, we enhanced the computing portal framework with container (virtual execution environment) technologies. In the original computing portal framework, software developers are able to publish their applications installed in HPC systems, but the installation of the applications have to be performed in a conventional manner. That is, the software developers have to copy and install their binary executables by themselves. In addition, they may have to install additional software/libraries that are required by their own programs, but it is sometimes difficult and/or even impossible because the administrators of the HPC systems usually do not allow the software developers to install such the software/libraries arbitrary. Another approach of installing software is to copy and build the binary executables from their source code, but it is sometimes troublesome and messy.

To address the abovementioned problem of installing software in HPC systems, we utilize container (virtual execution environment) technologies. A container is a kind of lightweight virtual execution environment that is isolated from its host environment and other containers. In other words, in a container, users are able to freely modify the environment of the container, that is, system administrators can let the users install any software they need without compromising security/safety of their systems, in theory.

More specifically, we utilized and integrated Docker (`http://docker.io`), a container system built on the Linux kernel, with our computing portal framework. In the computing portal framework extended with Docker, software developers are able to download a Docker container image that contains a basic execution environment of a HPC system, freely modify the image (i.e., install software/libraries) in order to prepare the execution environment required to run their applications, install their applications, and upload back the image to the computing portal framework. When publishing the applications, the software developers are able to specify the uploaded container images to be instantiated when the applications are launched as jobs. Moreover, the software developers are able to publish not only their applications, but also their container images so that other software developers can use the images.

In FY2015, we further enhanced the computing portal framework so that the users of the K computer are able to build and publish their own computing portal with their own authority and computing resources on the K computer.

In FY 2016, we further improved the implementation. Especially, we improved its compatibility and interoperability with computing systems other than the K computer, and designed/implemented a point system that is able to manage computing resources among users. In addition, we demonstrated the implementation at the open house of AICS.

4.3.2 Virtualization Techniques

4.3.2.1 Lightweight Virtualization for Testing/Debugging Parallel Programs

In order to utilize the full power of todays HPC systems as the K computer, users have to write massively parallel programs. However, writing parallel programs is difficult compared to conventional sequential programming. This is because parallel programs have inherent non-determinacy (e.g., process/thread execution order), that is, even if a parallel program contains a bug, it is not always easy to reproduce the bug. In addition, performance bottlenecks of parallel programs are not apparent from their source code because network latency, synchronization costs, scalability, etc. cannot be inferred directly from the source code. One possible solution to the problems is to utilize static source code analysis and dynamic performance profiling, however, some kind of bugs arise only when the number of processes/threads used by parallel programs is huge (e.g., several tens of thousands or more).

In order to address the abovementioned problem, in FY 2012, we designed a lightweight network virtualization technique that is useful for testing/debugging parallel programs. More specifically, we designed a virtualization framework that is able to provide several tens to hundreds of virtual execution environments per real (physical) execution environment. With the virtualization framework, users can test, debug, and/or profile their parallel programs on a limited number of physical computing nodes as if the programs run on a huge number of nodes.

The key of our virtualization framework is to virtualize network related operations at the level of shared libraries. Current popular virtualization technologies adopt virtualization at the level of CPU (with hardware assists) or OS (system calls), which is heavier than the level of shared libraries. This is because main purpose of the popular virtualization technologies is to provide virtual execution environments that are hard to be distinguished from real (physical) ones. Our virtualization framework, on the other hand, gives up to provide

such a realistic virtual execution environment, but instead aims to provide as much as possible of virtual execution environments per real (physical) one. By adopting virtualization at the level of shared libraries, the performance overheads of hooking system calls and/or CPU events (e.g., interrupts and exceptions) can be eliminated (while statically linked programs, that is, programs do not rely on shared libraries, will not be virtualized correctly).

In addition, our virtualization framework tries to reduce (or eliminate) exchange of virtual network routing information among real (physical) nodes. In order to correctly route packets from one virtual execution environment to another, all the physical nodes have to share the routing information of the virtual networks because the virtual execution environments may reside in different physical nodes. Therefore, if a single physical node manages the routing information, the node will become a performance bottleneck because all the other physical nodes have to synchronize with the node each time they need to route packets.

To address the problem, our virtualization framework tries to distribute the routing information statically (that is, before executing programs in virtual execution environments) as much as possible. In addition, even if dynamic updating of the routing information is inevitable, our virtualization framework tries to minimize synchronization between multiple physical nodes by separating allocation pool of physical (real) network ports statically.

In FY2013, we have implemented a prototype of our lightweight virtualization system based on the design of FY2012. Although there still remained bugs, it successfully ran on conventional PC clusters and Fujitsus FX10. More specifically, several MPI applications (including some of the NAS parallel benchmarks (NPB)) ran on our prototype virtualization system. In addition, we also ran Scalasca (a network performance profiling tool) on our system.

In FY2014 and FY2015, we improved the prototype of our lightweight virtualization system and it successfully ran on the K computer. More specifically, it successfully ran 20000 virtual computing nodes on 1000 physical computing nodes. Because the operating system kernel of the computing nodes of the K computer has a serious fault which is related to memory management, we could not increase the number of virtual computing nodes at that time.

In FY2016, we further improved the implementation in order to work around the abovementioned bug, and it successfully ran on 40000 virtual computing nodes on 2000 physical computing nodes of the K computer. In theory, it must be able to run more virtual computing nodes on a single physical computing node and run on more physical computing nodes, but this is not possible so far because the K computer restricts the number of user processes on a physical computing node.

4.3.2.2 Container Technologies for HPC

Container technologies are a kind of lightweight virtualization technology. Although they tend to be less efficient than the library-hooking approach described in the previous section, they provide more complete image of virtual execution environments. For example, Docker (`http://docker.io`) provides multiple isolated virtual Linux execution environments on a host Linux system. Because Docker is built and depends on several functionalities provided by the Linux kernel, it is not able to host non-Linux virtual execution environments unlike full-virtualization technologies (e.g., KVM, QEMU, and so on), but far more efficient than them.

One big problem of the current typical HPC systems compared to todays so-called cloud services from viewpoint of software developers/publishers is that the HPC systems are less flexible and/or responsive. For example, they are not allowed to install and/or modify system/middleware programs in the HPC systems, while the cloud services provide fully-virtualized environments to them and they can freely modify the environments. In addition, the typical HPC systems are operated with conventional batch schedulers and it sometimes takes time to launch jobs, while the cloud services launch virtual execution environments instantly when requested by them.

The reason why the conventional HPC systems are less flexible and/or responsive is that their primary purpose is to compute scientific applications efficiently as much as possible, thus the overheads that may be introduced by utilizing full virtualization technologies are unacceptable.

On the other hand, as described above, the recent advance in the container technologies achieves very small overheads yet provides sufficiently flexible virtual execution environments, thus we predict that the container technologies will play important role in forthcoming HPC usage.

Based on the abovementioned perspective, we are studying the possibilities of applying the container technologies (especially, Docker) to the HPC systems. More specifically, in FY2013 and FY2014, we developed dockerIaaSTools (`https://github.com/pyotr777/dockerIaaSTools`), which enables us to easily setup isolated multiple virtual execution environments to which users are able to login via SSH. In addition, as an application

of dockerIaaSTools, we extended K-scope (`http://www.aics.riken.jp/ungi/soft/kscope/`), which is a Fortran source code analysis tool developed by Software Development team of AICS, so that users are able to use the backend of K-scope that is installed in the remote server seamlessly as if it is installed in their local computers. Moreover, we also studied the internals of Docker and developed extensions that enable us to conserve storage for storing images containers (e.g., `https://github.com/pyotr777/docker-registry-driver-git`). Furthermore, as described above, we integrated Docker with our computing portal framework.

In FY2015, we also utilized Docker to improve the usability of K-scope. More specifically, we created a Docker container image in which K-scope is installed so that users are able to use K-scope without manually installing it. In addition, we also extended K-scope so that users are able to analyze their programs seamlessly on the remote server without modifying their source code and/or build scripts.

4.3.3 Program Verification and Analysis

4.3.3.1 Software Model Checking for Partitioned Global Address Space Language

Partitioned Global Address Space Languages (or, PGAS languages) are programming languages for distributed computing systems where the systems consist of large number of computing nodes and their memories are distributed among the nodes. In the PGAS languages, all the processes and/or threads in a program can share a single address space even though the memories are distributed, as in traditional distributed shared memory (DSM) systems. One of the distinguishing features of the PGAS languages is that the shared address space can be partitioned into sub-spaces and they can be bound to a specific process and/or thread explicitly. Thus, programmers can write a locality-aware program that is essential to achieve high performance on massively-parallel distributed memory systems of today (and future).

Despite the abovementioned advantage, one big problem with PGAS languages is that programmers can easily introduce concurrency bugs. For example, if multiple threads access a portion of a single address space simultaneously without proper synchronizations, race condition bugs can be easily introduced even if the accessed portion is bound to a specific process and/or thread. To make things worse, introducing synchronizations is not as easy as it sounds because excessive use of synchronizations severely degrades performance, while lack of them introduces hard-to-debug and non-reproducible concurrency bugs.

To address the problem, in FY 2012, we proposed and implemented a software model checking framework for PGAS languages. Software model checking is a program verification approach which tries to prove that a given program satisfies a certain property by exploring all the program states that can be reached during program execution. One problem of model checking PGAS programs is that it 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 they can dramatically reduce the number of states to be explored. However, it is not always easy to automatically infer proper abstractions because programs and properties to be verified vary.

To address the state explosion 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 to translate the trees. We also implemented CAF-SPIN, our proof-of-concept prototype of a model checking tool for Coarray Fortran. The experimental results with CAF-SPIN showed that abstractions can be defined easily and concisely by users, and the number of states to be explored for model checking is dramatically reduced with the abstractions.

Moreover, we also implemented XMP-SPIN, our software model checking tool for XcalableMP (`http://www.xcalablemp.org/`), and conducted several experiments with XMP-SPIN. More specifically, we conducted model checking of a number of parallel stencil computations written in XcalableMP (from small test programs to large application programs). Although stencil computations offer a simple and powerful programming style in parallel programming, they are sometimes error prone when considering optimization and parallelization because optimization of stencil computation may involve complex loop transformations and/or array reindexing, and parallelization requires explicit communication (data synchronizations) among multiple processes. In the experiments with XMP-SPIN, we checked whether there are no missing or redundant data synchronizations in the target programs, and successfully found four bugs in a reasonable time with a reasonable amount of memory.

4.3.3.2 Memory Consistency Model-Aware Program Verification

A memory consistency model is a formal model that specifies the behavior of the shared memory that is simultaneously accessed by multiple threads and/or processes. The recent multicore CPU architectures and shared memory multithread/distributed programming languages (e.g., Java, C++, UPC, Coarray Fortran, and so on) adopt relaxed memory consistency models. Under the relaxed memory consistency models, the shared memory sometimes behaves very differently from non-relaxed, sequential memory consistency models. For example, under some relaxed memory consistency models, the effects of the memory operations (e.g., A $\rightarrow$ B) performed sequentially by one thread may be observed in a different order (e.g., B $\rightarrow$ A) by the other threads. In addition, the threads may not agree on the observation orders of the effects of the memory operations (e.g., one thread observes A $\rightarrow$ B, while the other observes B $\rightarrow$ A, and so on). The reason why the recent CPUs and shared memory languages adopt relaxed memory consistency models is that a large number of threads and/or nodes share a single address memory space, thus enforcing non-relaxed, sequential memory consistency incurs huge synchronization overheads among the threads/nodes.

From the viewpoint of program verification, there are two problems in handling relaxed memory consistency models. First problem is that the conventional program verification approaches do not consider relaxed memory consistency models. Thus, they cannot be applied to relaxed memory consistency models because they may yield false results. Second problem is that there exist various kinds of relaxed memory consistency models and each CPU architecture/each programming language adopts different memory consistency models from each other. Therefore, it is tedious to define and implement a program verification approach for each CPU and programming languages of relaxed memory consistency models.

To address the problem, in FY2013, we studied three approaches. First approach is to define a new formal system that is able to represent various relaxed memory consistency models. More specifically, we define a very relaxed memory consistency model as a base model. On top of the base model, we defined various memory consistency models as additional axioms. With our formal system, we are able to define a broad range of memory consistency models from CPUs to shared-memory programming languages (e.g, Intel64, Itanium, UPC, Coarray Fortran, and so on), in the single formal system. With our formal system, we were able to proof the correctness of Dekkers mutual exclusion algorithm under the memory consistency model of Itanium.

Second approach is to design and implement a model checker that supports various relaxed memory consistency models based on the formal model of the first approach. More specifically, we define a non-deterministic state transition system with execution traces where each execution trace represents a possible permutation of instruction executions. Roughly speaking, given a target program, our model checker explores all the reachable states in the non-deterministic transition system of the target problem for all the possible execution traces (that is, permutations of instructions). In our model checker, memory consistency models can be defined as constraint rules on execution traces. For example, the sequential consistency model can be defined as a constraint that allows no permutation on the execution traces. With our model checker, we were able to verify the small examples programs of the specification manuals of the memory consistency models of Itanium and UPC. In addition, we were also able to formally discuss comparison of the two memory consistency models (Itanium and UPC).

Third approach is to define a new Hoare-style logic for a shared-memory parallel process calculus under a relaxed memory consistency model. More specifically, we define an operational semantics for the process calculus. Then define a sound (and relatively-complete) logic to the semantics. There are two key ideas in our Hoare-style logic. First idea is that a program is translated into a dependence graph among instructions in the program, and the operational semantics and the logic are defined in terms of the dependence graph. One advantage of handling dependence graphs is that while loops, branch statements, and parallel composition of processes can be handled in a uniform way. In addition, another advantage is that multiple memory consistency models can be handled by adopting different translation approaches for each memory consistency model. Second idea is that we introduce auxiliary variables in the operational semantics that temporarily buffer the effects of memory operations. Based on our Hoare-style logic, we also implemented a prototype semi-automatic theorem prover.

In FY2014, we optimized the implementation of our model checker (McSPIN) so that it can be applied to larger programs than the original implementation. More specifically, we introduced 4 optimization approaches: enhancing guard conditions, disabling speculation when unnecessary, prefetching instructions if possible, and removing the global time counter. In addition, in FY2014, we also enhanced our Hoare-style logic with a conventional rely-guarantee style rule in order to make the logic more compositional. More specifically, we added a new rely-guarantee style parallel composition rule because the original parallel composition rule is not compositional, that is, it requires us to infer all possible interleavings of parallel processes.

In FY2015, we further improved the implementation of McSPIN and studied the memory consistency model

of the programing language Chapel by request from a research developer of Chapel. Moreover, we also studied several memory management algorithms on various memory consistency models with external researchers.

In FY2016, we studied a unified and versatile theory for program analysis/verification under various relaxed memory consistency models, and enhanced/improved the implementation of McSPIN further. Especially, we used McSPIN for analyzing/verifying various concurrent copying garbage collection algorithms and various relaxed memory consistency models.

4.3.3.3 Evidence-Based Performance Tuning

In order to fully utilize the power of HPC systems, it is necessary to optimize and tune the performance of applications. However, performance tuning is a troublesome task because, even if performance bottlenecks/hotspots can be detected by performance profiling, it is not apparent how to rewrite programs to remove the bottlenecks/hotspots. In addition, generally speaking, modifying correctly working programs is reluctant from the viewpoint of developers. Thus, performance tuning requires experienced craftsmanship, and relies on intuition and experience.

In order to address the problem, we are working on an idea of evidence-based performance tuning. More specifically, we store the results of performance profiling in a database where the results are associated with source code modification history. With the database, developers are able to know, for example, what kinds of optimization were applied in the past, what kinds of optimization are effective for improving a certain performance profiling parameter, and so on. In FY2013, we conducted a preliminary experiment to implement the database and obtained promising results.

In FY2014, we developed a code mining mechanism that finds optimization patterns from source code modification history. More specifically, it calculates differences before and after modification at the level of abstract syntax trees and stores them to database. Then, we are able to search optimization patterns by searching database by queries that represent the patterns. More concretely, we defined about 40 queries that include loop unrolling, loop fusion, loop fission, loop interchange, array merging, array dimension interchange, code hoisting, and so on. In addition, we also created a so-called tuning catalog, which itemizes very small example programs that represents various optimization patterns for reference data. With the tuning catalog and several real tuning histories, we conducted a supervised learning (which is one of machine learning approaches) in order to suggest appropriate optimization approaches for a given source code and performance profiling data. More specifically, we solved a multi-label classification problem by translating it to multiple single-label classification problems with the binary relevance method and solving them with the k-NN algorithm. As feature vectors, we used the values of performance profiling data (e.g., cache-miss rate) and source code metrics (e.g., max loop depth). With an experiment with 469 tuning cases, we obtained satisfactory results, but the experiment was still too small to determine effectiveness of our approach.

In FY2015, we tried to increase the number of tuning cases in order to conduct detailed evaluation and improve accuracy of the analysis, but it turned out that it is hard to collect data directly because we could not find any researcher/developers who have such the data in and out of AICS. To work around the problem, we studied an approach of predicting performance of programs only from their source code modification history.

In FY2016, we further pursued the abovementioned approach of FY2015. More specifically, we designed and implemented an approach which is able to associate source code modification history with estimated performance data obtained by utilizing a B/F prediction approach without performing actual performance profiling. In fact, we analyzed several thousands of Fortran projects registered in GitHub (`https://github.com`), and were able to know their estimated performance characteristics.

4.3.3.4 Python-Based Aggregation of Multiple Software for HPC

In the world of HPC, programs are usually written in somewhat old-fashioned programing languages such as Fortran/C/C++ for historical reasons, thus writing programs for HPC is painful because we cannot use useful features of modern sophisticated programming languages. On the other hand, it is not realistic so far to write a whole program in modern programing languages because of performance problems.

In order to address the problem and achieve both productivity of program development and performance of program execution, we studied an approach of using Python for writing HPC applications. More specifically, we write non-performance critical large part of a program in Python, and performance critical small part in Fortran/C/C++. The reason why we choose Python is that Python provides a rich set of foreign language interfaces. For example, Fortran programs can be interfaced with f2py (NumPy: `http://www.numpy.org/`), C

programs can be interfaced with ctypes and Cython, and C++ programs can be interfaced with Boost.Python and Cython.

In FY2013, we modified EigenExa (a high-performance Eigen-solver developed by the Large-scale Parallel Numerical Computing Technology research team of AICS) so that it can be used as a shared library and a Python module (these modifications were feedbacked to the upstream). In addition, integration of Lotus (a quantum chemistry library developed by Dr. Tomomi Shimazaki, the Computational Molecular Science research team of AICS) and EigenExa were ongoing mainly by Tomomi Shimazaki.

In FY2014, in collaboration with Dr. Tomomi Shimazaki, a non-performance critical large part of Lotus was refactored and written in Python. We were able to utilize various existing libraries (e.g., EigenExa: `http://www.aics.riken.jp/labs/lpnctrt/EigenExa_e.html`, SMASH: `http://smash-qc.sourceforge.net/`, ASE: `https://wiki.fysik.dtu.dk/ase/`, etc.) in Lotus with the refactoring, and demonstrated that the features of Lotus can be easily extended. More specifically, we extended Lotus by request of Yukio Kawashima of the Computational Chemistry research unit of AICS with only several tens of lines of code addition.

In FY2015 and FY2016, we further refactored Lotus by using Cython, and our Python approach has been practiced by Dr. Kazuo Kitaura for realizing his new quantum chemistry calculation theory, with the help of Dr. Tomomi Shimazaki. In addition, we examined an approach of constructing a database that stores the results of ab initio calculations for molecules of PubChem (a famous chemical database of molecules), and applying machine learning and data mining approaches on the database. More specifically, we conducted several preliminary experiments utilizing machine learning approaches and performance evaluation of elemental technologies for constructing the database.

4.3.3.5 Porting Performance Analysis Tools to the K computer

Because massively parallel supercomputers, such as the K computer, are very different from single computer systems or small size cluster systems, simply porting existing applications to the K computer typically does not work due to performance problems (many existing conventional applications do not consider massively-parallel systems). Therefore, performance profiling is necessary to understand the behaviors of applications on massively parallel systems and tune the applications.

To address the problem, we are porting/deploying existing performance analysis tools to the K computer, in cooperation with external research institutes. More specifically, in FY2014, we ported two performance analysis tools to the K computer: Scalasca (`http://www.scalasca.org/`) and Extrae (`https://www.bsc.es/computer-sciences/extrae`). Scalasca was ported in cooperation with a research team of Juelich Supercomputing Centre, and Extrae was ported in cooperation with a research team of Barcelona Supercomputing Center. Using the ported tools, we actually analyzed the behavior of ABySS (`http://www.bcgsc.ca/platform/bioinfo/software/abyss`), a parallel genome sequence assembler. We also analyzed the behavior of SIONlib, a parallel I/O library, in cooperation with J 端 lich Supercomputing Centre, and deployed it on the K computer.

In FY2015, we continued to port/deploy existing performance analysis tools to the K computer. Especially, we ported Eclipse PTP (`https://eclipse.org/ptp/`), which is an extension framework of Eclipse (`https://eclipse.org/`) for parallel program development/execution, to the K computer, in cooperation with a research team of University of Oregon. In addition, we modified and integrated Extrae and SIONlib so that Extrae is able to use SIONlib for its I/O processing (this should be useful for handling very large trace data).

In FY2016, we further improved the ported implementation of Extrae for the K computer in cooperation with a research tea of Barcelona Supercomputing Center, and published it on the K computer. In addition, we conducted performance analysis of several applications on the K computer. For example, we analyzed the performance of the brain simulator NEST, and published the results of the analysis as a reviewed journal article. We also ported and analyzed libraries for machine learning (deep-learning, especially) on the K computer.

4.4 Schedule and Future Plan (FY2017 ∼)

From FY2017, the HPC Usability Research Team aims to increase the number of applications of the K computer. It especially focuses on applications that make use of both simulation and data analysis. We expect that the combination of simulation and data analysis will increase the value of both components because data can improve simulation accuracy and simulations can generate valuable data. However, developing such combined applications is difficult because developers must have expertise in computer systems, as well as simulation and data analysis algorithms, to connect the different types of programs. Our team studies tools and frameworks

that simplify the process of developing and executing such combined applications so that more people, especially those in industry, can use the supercomputer for their innovative products and services.

More specifically, we will study software for creating data analysis workflow and framework for combining data analysis and numerical calculation.

4.5 Publications (FY2012 ∼ FY2016)

4.5.1 Journal Articles

[1] Abe, T. and Maeda, T.: "Concurrent Program Logic for Relaxed Memory Consistency Models with Dependencies across Loop Iterations", Journal of Information Processing 25:244-255. Jan. 2017.
https://doi.org/10.2197/ipsjjip.25.244

[2] Hahne, J., Helias, M., Kunkel, S., Igarashi, J., Kitayama, I., Wylie, B., Bolten, M., Frommer, A., and Diesmann, M.: "Including Gap Junctions into Distributed Neuronal Network Simulations", In: Brain Inspired Computing, eds: Katrin Amunts, Lucio Grandinetti, Thomas Lippert, Nicolai Petkov. Lecture Notes in Computer Science 10087, Springer, 43-57. Dec. 2016.
https://doi.org/10.1007/978-3-319-50862-7_4

[3] Abe, T. and Maeda, T.: "A General Model Checking Framework for Various Memory Consistency Models", International Journal on Software Tools for Technology Transfer (2016). Jul. 2016.
https://doi.org/10.1007/s10009-016-0429-y

4.5.2 Conference Papers

[4] Hashimoto, M., Terai, M., Maeda, T, and Minami, K.: "An Empirical Study of Computation-Intensive Loops for Identifying and Classifying Loop Kernels", In Proceedings of the 8th International Conference on Performance Engineering (ICPE 2017), pp. 361-372, Apr. 2017.
https://doi.org/10.1145/3030207.3030217

[5] Abe, T. and Maeda, T.: "Observation-Based Concurrent Program Logic for Relaxed Memory Consistency Models", In Proceedings of the 14th Asian Symposium on Programming Languages and Systems (APLAS 2016), LNCS 10017, pp. 63-84. Dec. 2016.
https://doi.org/10.1007/978-3-319-47958-3_4

[6] Abe, T. and Maeda, T.: "Reducing State Explosion for Software Model Checking with Relaxed Memory Consistency Models", In Proceedings of Symposium on Dependable Software Engineering (SETTA 2016), LNCS 9984, pp. 118-135. Nov. 2016.
https://doi.org/10.1007/978-3-319-47677-3_8

[7] Shimazaki, T., Hashimoto, M., and Maeda, T.: "Developing a High Performance Quantum Chemistry Program with a Dynamic Scripting Language", In Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SEHPCCSE15), online, Austin, TX, USA, Nov. 20, 2015.
http://doi.org/10.1145/2830168.2830170

[8] Kitayama, I., Wylie, B. J. N, and Maeda, T.: "Execution Performance Analysis of the ABySS Genome Sequence Assembler using Scalasca on the K computer", In Proceedings of ParCo2015, Edinburgh, UK, Sep. 2015.
https://doi.org/10.3233/978-1-61499-621-7-63

[9] Abe, T. and Maeda, T.: "Towards a Unified Verification Theory for Various Memory Consistency Models", In Proceedings of the 6th Workshop on Syntax and Semantics of Low-Level Languages (LOLA 2015), Short Paper, online, Kyoto, Japan, Jul. 5, 2015.

[10] Hashimoto, M., Terai, M., Maeda, T., and Minami, K.: "Extracting Facts from Performance Tuning History of Scientific Applications for Predicting Effective Optimization Patterns", In Proceedings of the 12th Working Conference on Mining Software Repositories (MSR 2015), pp. 13-23, Florence, Italy, May 16, 2015.
https://doi.org/10.1109/MSR.2015.9

[11] Abe, T. and Maeda, T.: "Optimization of a General Model Checking Framework for Various Memory Consistency Models", In Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS 2014), pp. 14:1-14:10, Eugene, OR, USA, Oct. 10, 2014.
https://doi.org/10.1145/2676870.2676878

[12] Terai, M., Bryzgalov, P., Maeda, T., and Minami, K.: "Extending K-scope Fortran Source Code Analyzer with Visualization of Performance Profiling Data and Remote Parsing of Source Code", In Proceedings of the 6th International Symposium on Advances of High Performance Computing and Networking (AHPCN) within International Conference on High Performance Computing and Communications (HPCC-2014), pp. 878-885, Paris, France, Aug. 22, 2014.
https://doi.org/10.1109/HPCC.2014.149

[13] Abe, T. and Maeda, T.: "A General Model Checking Framework for Various Memory Consistency Models", In Proceedings of the 19th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2014), pp. 332-341, Phoenix, AZ, USA, May 19, 2014.
https://doi.org/10.1007/s10009-016-0429-y

[14] Abe, T. and Maeda, T.: "Model Checking with User-Definable Memory Consistency Models", In Proceedings of the 7th Conference on Partitioned Global Address Space Programming Models (PGAS 2013), Short paper, online, Edinburgh, UK, Oct. 4, 2013.

[15] Abe, T., Maeda, T., and Sato, M.: "Model Checking Stencil Computations Written in a Partitioned Global Address Space Language", In Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2013). pp. 365-374, Cambridge, MA, USA, May 20, 2013.
https://doi.org/10.1109/IPDPSW.2013.90

[16] Abe, T., Maeda, T., and Sato, M.: "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 2012), online, Santa Barbara, CA, USA, Oct. 12, 2012.

4.5.3 Posters and Presentations

4.5.3.1 Posters

[17] Abe, T. and Maeda, T: "Research on Program Verification Considering Various Memory Consistency Models", The 13rd Dependable System Workshop (DSW2015). Poster. 2015. In Japanese.

[18] RIKEN AICS HPC Usability Research Team: "HPC Usabiilty Research Team", Exhibition of International Supercomputing Conference (ISC) High Performance. Poster. 2015.

[19] Bryzgalov, P.: "Eclipse PTP and TAU with K and FX10 Supercomputers", Exhibition of International Supercomputing Conference (ISC) High Performance. Flyer. 2015.

4.5.3.2 Invited Talks

[20] Maeda, T.: "Trial Study on Development of a Quantum Chemistry Library with Modern Programming Techniques", The 1st Workshop on High Performance Computing Chemistry (HPCC 2015), Kobe, Japan, Dec. 1, 2015. In Japanese.

[21] Hashimoto, M.: "Towards Evidence-based Performance Tuning Assist", The 6th Symposium on Automatic Tuning Technology and its Application (ATTA 2014). 2014. In Japanese.

4.5.3.3 Presentations

[22] Kitayama, I.: "DynInst on arm64 – Status", Linaro Connect Budapest 2017.

[23] Abe, T.: "Compositional Concurrent Program Logic for Relaxed Memory Consistency Models", Workshop on Computer Science and Category Theory (CSCAT 2015). 2015. In Japanese.

[24] Kitayama, I.: "Parallel File I/O Optimization with SIONlib", Japan Lustre Users Group 2014 (JLUG 2014). 2014.

[25] Hashimoto, M.: "Constructing Finge-Grained Tuning Cases Database and Its Application for Prediction of Effective Program Optimizations", The 10th Autotuning Research Group's Open Academic Session (ATOS10). 2014. In Japanese.

[26] Abe, T.: "Towards Semi-automatic Theorem Proving Considering Memory Consistency Models", The 25th Algebra, Logic, Geometry and Informatics (ALGI). 2014. In Japanese.

[27] Abe, T.: "Program Verification for Formalized Relaxed Memory Consistency Models", The 31st Symbolic Logic and Computer Science (SLACS). 2014. In Japanese.

[28] Kitayama, I.: "A User's Experience with FEFS", In Japan Lustre User Group 2013. 2013.

[29] Maeda, T.: "Brief Introduction of HPC Usability Research Team", In the 4th AICS International Symposium. 2013.

4.5.4 Patents and Deliverables

4.5.4.1 Patents

[30] "Program analysis/verification service provision system, control method for same, computer readable non-transitory storage medium, program analysis/verification device, program analysis/verification tool management device", Assignee: Japan Science and Technology Agency, Inventor: Toshiyuki Maeda, US Patent No.9400887, Japan Patent No.5540160, and so forth.

4.5.4.2 Deliverables

[31] CCA/EBT: Code Comprehension Assistance for Evidence-Based performance Tuning, `https://github.com/ebt-hpc/cca`, 2017∼

[32] Eclipse PTP ported for the K computer, `https://github.com/pyotr777/EclipsePTP_PJM_TSC/`, 2015∼

[33] McSPIN. `https://bitbucket.org/abet/mcspin`, 2014∼

[34] Python ported to the K computer. Available on the K computer. Simply do ". /opt/aics/hpcu/env.sh" on the computing nodes of the K computer, then follow the documents of Python, 2014∼

[35] NumPy ported to the K computer. Available on the K computer. Simply do ". /opt/aics/hpcu/env.sh" on the computing nodes of the K computer, then follow the documents of NumPy, 2014∼

[36] MPI4Py library ported to the K computer. Available on the K computer. Simply do ". /opt/aics/hpcu/env.sh" on the computing nodes of the K computer, then follow the documents of MPI4Py, 2014∼

[37] MapReduce-MPI library ported to the K computer. Available on the K computer. Simply do ". /opt/aics/hpcu/env.sh" on the computing nodes of the K computer, then follow the documents of MapReduce-MPI, 2014∼

[38] Python binding of Rokko (Integrated Interface for Libraries of Eigenvalue Decomposition) (joint work with Dr. Sakashita and Prof. Todo of the University of Tokyo. Feedbacked to the original source tree of Rokko), 2014∼

[39] TAU ported to the K computer. Available on the K computer, 2013∼

[40] Extrae ported to the K computer. Available on the K computer, 2013∼

[41] Scalasca ported to the K computer. Available on the K computer (Joint work with Programming Environment Research Team of AICS), 2013∼

[42] Python binding of EigenExa (joint work with Dr. Shimazaki of Computational Molecular Science Research Team of AICS. Partially feedbacked to the original source tree of EigenExa, developed by Large-scale Parallel Numerical Computing Technology Research Team), 2013∼

[43] DockerIaaStool: Tools for creating a simple Infrastructure-as-a-Service system with Docker, 2013∼

[44] K-scope with SSHConnect (joint work with Software Development team of AICS), 2013∼