HPC Usability Research Team

1. Team Members

Toshiyuki Maeda (Team Leader)

Masatomo Hashimoto (Research Scientist)

Tatsuya Abe (Research Scientist)

Petr Bryzgalov (Research Scientist)

Itaru Kitayama (Technical Staff I)

Yves Caniou (Visiting Scientist, University of Tokyo)

Yoshiki Nishikawa (Visiting Scientist, University of Tokyo)

Sachiko Kikumoto (Assistant)

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:

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.

2. **Virtualization**

   Virtualization is a technology for realizing virtual computers on real (physical) computers. One big problem of the above mentioned 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.

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 above mentioned 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.

3. Research Results and Achievements

3.1. Design and Implementation of a Computing Portal Framework for HPC

Based on the prototype design and implementation of a computing portal framework in 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 our computing portal framework 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.

One big limitation of our current computing portal framework is that security mechanisms are still not introduced. That is, (maybe malicious) users and/or applications can easily access the other users/applications data. This problem has been already recognized in FY2012, but we have not addressed the problem directly in our prototype framework. Instead, in FY2013, we studied so-called container (or sandbox) mechanisms, that enable users to isolate their computing environments from other users. As a first step, we investigated a lightweight container system called Docker (https://www.docker.io/), and implemented a utility tool which is able to give every user an isolated computing environment in the form of a container of Docker.

In FY2014, we will continue to develop our computing portal framework. Especially, we plan to integrate some kind of security mechanism (e.g., Docker, if possible) to our framework. We also plan to integrate our framework to the real K computer, but we recognize that it is not so easy from the viewpoint of the operation policy of the K computer.

3.2. Virtualization Techniques

1. **Lightweight virtualization for testing/debugging parallel programs**

   Writing a program for massively parallel HPC environments (e.g., K computer) is a hard task. This is mainly because parallel programs inherently have non-determinacy, thus, it is sometimes extremely difficult to debug a bug in parallel programs, because the bug may not be easily reproducible. In addition to the hard-to-debug problem, there is also a performance problem in writing massively parallel programs. It is not uncommon that, even if a program scales on a PC cluster system whose size is small-to-moderate, the performance of the program severely degrades on massively parallel HPC systems. This is because communication costs between computing nodes may largely vary and sometimes incurs unacceptable heavy overheads.

   In order to address the problem, we have been developing a lightweight network virtualization system for testing/debugging programs for massively parallel programs without actually using real massively parallel HPC environments. With our system, users can run several hundreds of virtual computing nodes on a single physical computing node.

   One key idea of our lightweight virtualization system is to utilize the library-hooking approach, that is, intercept function calls for network related operations from user programs, and modify parameters and/or return values of the function calls in order to "trick" the user programs as if they are executed in isolated virtual computing nodes, even

though they run on a single computing node. One benefit of the library-hooking approach is that it introduces little overheads to program execution (compared to other virtualization techniques, e.g., CPU level virtualization, OS level virtualization, and so on) because it can be achieved by user-level operations only and requires no interaction with OS.

Another key idea of our lightweight virtualization system is reduction of the costs of network routing management by statically distributing routing information as much as possible. The routing information is necessary to correctly route network packets from one virtual node to another where they may reside in different physical computing node. Therefore, if the routing information is maintained by one single physical node, all the physical nodes have to communicate with the manager node each time they need to route packets from one virtual node to another, thus the node results in a performance bottleneck.

In order to address the problem, our lightweight virtualization system statically distributes the routing information as much as possible before executing user programs on virtual computing nodes. In ordinary HPC environments, the network topology of each job execution is fixed during the job execution. Therefore, our static distribution of the routing information should work in most cases. Even if some jobs require dynamic node allocation, our system tries to minimize the cost of updating the routing information by carefully allocating virtual network port. More concretely, we first divide the range of the available network ports into disjoint ranges, and allocate the divided range to each physical computing node.

Based on the above approaches, in FY2013, we have implemented a prototype of our lightweight virtualization system. Although there still remain bugs, it successfully runs on conventional PC clusters and Fujitsu's 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, we will continue the development of our system and plan to study an approach of tricking performance profiling tools so that they feel as if they run on real computing nodes and emit profiling data which represents characteristics of real massively-parallel computing environments.

2. CPU emulators for SPARC 64 VIIIfx

One big problem of the current K computer from the viewpoint of usability is that it adopted SPARC architecture (more precisely, SPARC 64 VIIIfx architecture), which is rarely used in ordinary PCs, servers, and HPC (they are almost dominated by Intel architectures). Therefore, in order to utilize the K computer, ordinary users have to prepare source code of applications that they want to run, cross-compile the source code on the front-end node of the K computer because binary executables for Intel architecture do not run on SPARC architecture directly. To make things worse, cross-compilation of applications sometimes produce malfunctioning executables partly because the applications do not consider CPU architectures but Intel architecture. Therefore, users have to test the cross-compiled executables on the K computer whether they work expectedly or not.

In order to address the problem, we are studying on CPU emulators. A CPU emulator is a program which emulates the effects of instructions of a CPU architecture. More specifically, we are working on two kinds of CPU emulators. One is a CPU emulator which emulates SPARC 64 VIIIfx on Intel architecture, and another is a CPU emulator which emulates Intel architecture on SPARC 64 VIIIfx. With the former SPARC 64 VIIIfx on Intel emulator, users are able to test the cross-compiled executables for the K computer on their own development environments (PC, clusters, and so on). With the latter Intel on SPARC 64 VIIIfx emulator, users are able to run their binary executables for Intel architecture on the K computer without cross-compiling their source code.

More concretely, we are developing the two emulators by extending the existing CPU emulator QEMU. For the SPARC 64 VIIIfx on Intel emulator, we extended QEMU to support the features specific to the SPARC 64 VIIIfx architecture (e.g., extended general purpose/floating-pointer registers, SIMD extension, and so on). For the Intel on SPARC 64 VIIIfx, we fixed bugs of QEMU that prevents normal operations of QEMU on the K computer.

In FY2013, we continued development of the SPARC 64 VIIIfx on Intel emulator from FY2012, but we could not complete the development (there still remain several severe bugs that prevent many applications from working). On the other hand, the prototype implementation of the Intel on SPARC VIIIfx emulator has been completed and several applications successfully run on the emulator. However, its performance is not satisfactory because its execution time is 10 to 20 times slower.

In FY2014, we will continue the development of the two CPU emulators. For the SPARC 64

VIIIfx on Intel emulator, we aim to implement a more stable system which is usable for practical testing of applications that should be executed on the K computer. For the Intel on SPARC 64 VIIIfx emulator, we will improve its execution performance.

3.3. Program verification and analysis

1. Memory Consistency Model-Aware Program Verification

A memory consistency model is a formal model which specifies the behavior of the shared memory which 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→B) performed sequentially by one thread may be observed in a different order (e.g., B→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→B, while the other observes B→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 which 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 Dekker's mutual exclusion algorithm under the memory consistency model of Itanium.

Second approach is to design and implement a model checker which 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 which 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.

2. 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. However, because the experiment was still very preliminary with very little number of application programs, we do not have full confidence, so far. In FY2014, we plan to conduct the more realistic experiment with larger number of applications with more realistic source code modification histories.

4. Schedule and Future Plan

In FY 2014, we will improve the prototype implementation of our computing portal. As mentioned above, the current implementation does not have rigid security mechanism. In order to address the security problem, we will modify and/or extend the current APIs/protocols of our computing portal and apply them to the implementation. In addition, we will also consider integrating a security sandbox system (e.g., Docker) to our implementation. Besides the security problem, we also plan to integrate our framework to the real K computer, if our security and political policies allow.

Regarding the virtualization technologies, we will continue to implement the lightweight network virtualization framework for testing/debugging parallel programs. Especially, we will design and implement a mechanism which tricks performance profiling tools so that they feel as if they run on real computing nodes and emit profiling data which represents characteristics of real massively parallel computing environments. In addition, we will also continue to implement the SPARC 64 VIIIfx on Intel emulator and Intel on SPARC 64 VIIIfx emulators.

Regarding the program verification and analysis, we will conduct more experiments with our three approaches for program verification under relaxed memory consistency models to evaluate their effectiveness and practicality. In addition, we will also consider designing and

implementing a more simple and concise logic/model based on the experiences of the three approaches. Regarding the evidence-based performance tuning, we plan to conduct the more realistic experiment with larger number of applications with more realistic source code modification histories with the prototype implementation developed in FY2013.

In addition to the above mentioned individual research topics, we plan to start integrating the research results of the virtualization technologies and the software verification into the computing portal somewhere from the second half of FY 2014 to the first half of FY 2015.

5. Publication, Presentation and Deliverables

(1) Conference Papers

[1] Abe, T. and Maeda, T., "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).

[2] Abe, T. and Maeda, T., "Model Checking with User-Definable Memory Consistency Models", In Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models (PGAS 2013), Short paper, online.

[3] 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). To appear.

(2) Presentations

[4] Kitayama, I., "A User's Experience with FEFS", In Japan LUG (Lustre User Group) 2013.

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

(3) Software

[6] K-scope with SSHConnect (joint work with Software Development team of AICS):
URL: http://www.kcomputer.jp/ungi/soft/kscope/

[7] Python binding of EigenExa (joint work with Dr. Shimazaki of Computational Molecular Science Research Team of AICS. Partially feedbacked to the original developers of Large-scale Parallel Numerical Computing Technology Research Team)

[8] DockerIaaStool: Tools for creating a simple Infrastructure-as-a-Service system with Docker
URL: https://github.com/pyotr777/dockerIaaSTools

[9] QEMU on the K computer: CPU emulator for executing Intel binary executables on the K computer (implemented by extending the original QEMU emulator. In preparation for publication.)

[10] ABySS on the K computer: Parallel, paired-end sequence assembler (implemented by modifying the original ABySS so that it scales on the K computer. In preparation for publication.)