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)
Judit Gimenez (Visiting Scientist, Barcelona Supercomputing Center)
Sameer Shende (Visiting Scientist, University of Oregon)
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

In FY2013, we developed a web-based user-interface for the computing portal framework developed in FY2012-FY2013. With the web interface, software developers can easily publish their applications installed in HPC systems, and users are able to launch the published applications. The web-interface is designed to be flexible in the sense that it can be accessed by not only web browsers run on PCs, but also modern smartphones. Thus, users can launch and monitor their jobs through their smartphones, anywhere, anytime.

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.

From viewpoint of deploying the computing portal framework, there remain three problems. First problem is that the current implementation does not provide an accounting mechanism. That is, there is no way to know which application is executed by whom or who is to be charged for consumed computing resources. This will be even more complex when multiple applications (published by multiple software developers) and multiple users are involved. Second problem is that the container system of Docker cannot run on the computing nodes of the K computer (or FX10) because their Linux kernel version is too old to execute Docker properly (while its login nodes are new enough). The problem can be solved by upgrading the Linux kernel, but it is not easy because the Linux kernel running on the K computer is extensively modified in order to address problems caused in massively parallel computing environments. Third problem is the operation policy of the K computer. The primary purpose of the K computer is to perform scientific computation, thus its operation policy focuses on executing such the computation efficiently and reliably. Therefore, it is hard to directly integrate our computing portal framework with the K computer. Instead, we are currently using FX10 of AICS Research Division for experiments.

In FY2015, we will continue to develop our computing portal framework. Especially, we plan to implement some kind of accounting mechanisms to our framework. We also plan to upgrade the Linux kernel of the computing nodes (of our FX10 and/or the K computer), but we recognize that it is extremely difficult.

3.2. Virtualization Techniques

1. Lightweight virtualization for testing/debugging parallel programs

Writing a program which makes full use of massively parallel HPC environments (e.g., the K computer) is extremely difficult because debugging parallel programs is a hard task due to inherent non-determinacy of parallel programs and hard-to-reproduce bugs. Moreover, writing massively parallel programs also tend to suffer from a performance problem. For example, even if a program scales well on a PC cluster system whose size is small-to-moderate, the program may not scale on massively parallel HPC systems. Even worse, the performance may severely degrade and will be worse than on a small PC cluster system or even a single PC. Actually, this is not uncommon and the reason is that communication costs between computing nodes may largely vary and sometimes incurs unacceptable heavy overheads.

In order to address the abovementioned 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.

There are two key ideas in our system: library-hooking and decentralized management of routing information. Library-hooking is a kind of virtualization technology which intercepts function calls for system operations, and modify their parameters and/or return values in order to trick the programs as if they run on in isolated multiple computing nodes, even though they run on a single physical computing node. More specifically, in our lightweight virtualization system, we mainly hook network related operations (and some file I/O) from user programs. 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 and requires no interaction with OS.

When implementing a lightweight network virtualization system, decentralized management of routing information is necessary in order to avoid maintaining routing information in a single or a few physical nodes. Our lightweight virtualization system has to manage routing information by itself because it virtualizes network environments. If the routing information is managed in a single physical node, all the other physical nodes have to ask the single node in order to correctly route network packets from one virtual node to another. Therefore, when the numbers of virtual nodes and physical nodes are huge, the single node will become a performance bottleneck and severely degrade the overall performance of our lightweight virtualization system.

More specifically, our lightweight virtualization system statically distributes the information which virtual node runs on which physical node before executing user programs on virtual computing nodes. In ordinary HPC systems, it is uncommon that computing nodes are directly allocated during job execution. In addition, in order to virtualize port numbers, our lightweight virtualization system let physical nodes exchange the information about virtualized port numbers when one virtual node on one physical node communicates with another virtual node on another physical node. In our

previous implementation of FY2013, we also distributed the information of the virtual port numbers statically before executing user programs by dividing the range of the available physical ports into disjoint ranges and allocating the divided range to each physical node. However, we recognized that this approach does not scale when the number of the physical nodes become huge. Thus, we applied the dynamic exchange approach as described above.

Based on the abovementioned approaches, in FY2014, we have implemented a prototype of our lightweight virtualization system. It successfully runs on conventional PC clusters and Fujitsu's FX10. On the K computer, it successfully runs 20000 virtual computing nodes on 1000 physical computing nodes. 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 currently the K computer restricts the number of user processes on a physical computing node and the operating system kernel of the computing nodes of the K computer has a serious fault which is related to memory management.

In FY2015, we will continue the development of our system and evaluate it with more large numbers of virtual computing nodes and physical computing nodes. In addition, we 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. Container technologies for HPC

As slightly described in Sec. 3.1, 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 (Sec. 3.2.1), 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 today's 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 FY2014, we continued to develop 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.kcomputer.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 studying the internals of Docker and developed extensions that enable us to conserve storage space for storing/managing imaged of Docker containers (e.g., https://github.com/pyotr777/docker-registry-driver-git). Furthermore, as described above (in Sec. 3.1), we integrated Docker with our computing portal framework.
In FY2015, we will continue to study the possibility of applying the container technologies to the HPC systems. For example, we plan to study the possibility of running Docker on the computing nodes of the K computer (or FX10 of AICS Research Division), though it is difficult because the Linux kernel running on the computing nodes are too old to run Docker. We have to know (and decide) which is more practical, modify the Linux kernel of the K computer, or target the forthcoming next-generation super computer.

3.3. Program verification and analysis

1. Memory Consistency Model-Aware Program Verification

A memory consistency model is a formal model that specifies the behavior of the memory that is shared and simultaneously accessed by multiple threads and/or processes. Under the recent multicore CPU architectures and shared memory multithread/distributed programming languages (e.g., Java, C++, UPC, Coarray Fortran, and so on), the shared memory sometimes behaves in an unexpected way because they adopt *relaxed* memory consistency models. For example, under some relaxed memory consistency models, the effects of the memory operations performed sequentially by one thread (e.g., A→B) may be observed in a different order by the other threads (e.g., B→A). Moreover, the threads may not agree on the orders of the effects of the memory operations (e.g., one thread observes A→B, while the other observes B→A, and so on) they observe. The reason why the recent CPUs and shared memory languages adopt relaxed memory consistency models is that enforcing sequential (non-relaxed) memory consistency incurs huge synchronization overheads among a large

number of the threads/nodes that share a single address memory space.

From the viewpoint of program verification, there are two problems in handling relaxed memory consistency models. First problem is that the conventional program verification does not consider relaxed memory consistency models. Thus, they cannot be applied directly 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. Thus, it is very tedious to define and implement program verification for each CPU and programming languages of relaxed memory consistency models.

To address these problems, we have been studying 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. Then, we define various memory consistency models as additional axioms on the base model. In fact, 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.

Second approach is to design and implement a model checker that supports various relaxed memory consistency models based on the formal model of the abovementioned 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 examples programs of the specification manuals of the memory consistency models of 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, and 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.

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.

2. Evidence-Based Performance Tuning

To get the maximum of HPC systems, it is inevitable to optimize the performance of applications. However, performance tuning for massively parallel HPC systems is very difficult because it is not apparent how to improve programs except for highly skilled programmers. In addition, generally speaking, modifying correctly working programs is a bothering task from the viewpoint of developers. Thus, performance tuning requires experienced craftsmanship, and relies on intuition and experience.

In order to address the problem, we have been working on *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 FY2014, we developed a code mining mechanism which 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 enumerates 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 plan to conduct the more realistic experiment with larger number of applications by designing and implementing automatic collector of source code modification histories and their corresponding performance tuning data.

4. Schedule and Future Plan

In FY 2015, we will continue to improve the implementation of our computing portal. As described in Sec. 3.1, one problem of the current implementation is that it does not provide an accounting mechanism to know which application is executed by whom or who is to be charged for consumed computing resources. In order to address the problem, we plan to implement some kind of accounting mechanisms to our framework.

Regarding the virtualization technologies, we will continue to implement and evaluate the lightweight network virtualization framework for testing/debugging parallel programs. More specifically, we will evaluate our implementation on larger number of the physical computing nodes with larger number of the virtual computing nodes, on the K computer. In addition, we will also continue to design and implement a mechanism which tricks performance profiling tools so that they feel as if they run on physical computing nodes and emit profiling data which represents characteristics under the real parallel computing environments. As for the container technologies, we will continue to study the possibility of applying the container technologies to the HPC systems. In addition, we plan to pursue the possibility of upgrading the Linux kernel which runs on the computing nodes of our FX10 in order to fully leverage the power of the container technology (Docker).

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, and improve them. Regarding the evidence-based performance tuning, we plan to conduct a relatively larger-scale experiment with larger number of applications with more realistic source code modification histories to determine effectiveness of our approach. In addition, we also plan to design and implement automatic collector of source code modification histories and their corresponding performance tuning data.

In addition to the above mentioned individual research topics, we also plan to design/implement integration of the research results of the virtualization technologies and the software verification with the computing portal.

5. Publication, Presentation and Deliverables

(1) Conference Papers (Refereed)

1. 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, 2014.

2. Terai, M., Bryzgalov, P., Maeda, T., and Minami, K., "Extending Kscope Fortran Source Code Analyzer with Visualization of Peformance 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 Inrternational Conference on High Performance Computing and Communications (HPCC-2014), pp. 878-885, 2014.
3. 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, 2014.
4. Hashimoto, M., Mori, A., and Izumida, T., "A Comprehensive and Scalable Method for Analyzing Fine-grained Source Code Change Patterns", In Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015), pp. 351 - 360, 2015.
5. 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), To appear.

(2) Invited Talks

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

(3) Presentations

7. Abe, T., "Program Verification for Formalized Relaxed Memory Consistency Models", The 31st Symbolic Logic and Computer Science (SLACS), 2014. (In Japanese)
8. Abet, T., "Towards Semi-automatic Theorem Proving Considering Memory Consistency Models", The 25th Algebra, Logic, Geometry and Informatics (ALGI), 2014. (In Japanese)
9. 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)
10. Kitayama, I., "Parallel File I/O Optimization with SIONlib", Japan Lustre Users Group 2014 (JLUG 2014), 2014.
11. Abe, T., "Compositional Parallel Program Logic for Relaxed Memory Consistency Models",

Workshop on Computer Science and Category Theory (CSCAT 2015), 2015.

(4) Deliverables

12. Boost library (http://www.boost.org/) ported to the K computer. Available on the K computer.
13. MapReduce-MPI library (http://mapreduce.sandia.gov) ported to the K computer. Available on the K computer.
14. MPI4Py library (http://mpi4py.scipy.org/) ported to the K computer. Available on the K computer.
15. NumPy library (http://www.numpy.org/) ported to the K computer. Available on the K computer.
16. Python (https://www.python.org/) ported to the K computer. Available on the K computer.
17. A Python wrapper library for EigenExa (http://www.aics.riken.jp/labs/lpnctrt/EigenExa_e.html) (updated). Available on the K computer.