Chapter 1

System Software Research Team

1.1 Members

Yutaka Ishikawa (Team Leader)

Atsushi Hori (Senior Scientist)

Masamichi Takagi (Senior Scientist)

Balazs Gerofi (Research Scientist)

Takahiro Ogura (Research & Development Scientist)

1.2 Overview of Research Activities

The system software research has been conducted in cooperation with the System Software Development Team in the Flagship 2020 project. The team focuses on the research and development of an advanced system software stack not only for the "K" computer but also for toward exa-scale computing including Fugaku.

We have been mainly focusing on scalable high performance communication and file I/O libraries and/or middlewares. The former research topics, sharing virtual address space and IHK/McKernel light-weight kernel, have been almost taken over by the System Software Development Team, but the research results are shown here.

1.3 Research Results and Achievements

1.3.1 RIKEN-MPICH

We have been implementing an MPI library, based on MPICH developed by Argonne National Laboratory, for the Fugaku supercomputer under the DoE/MEXT collaboration in order to provide MPICH-based MPI for Fugaku.

The MPICH implementation has two layers, so-called MPID and device layers as shown in Figure 1.1. Basically, the MPID layer implements all the MPI functionalities using the device interface. The device layer CH4 is the latest device implementation. It consists of network and shared memory implementation layers, named "netmod" and "shmmod," respectively. The netmod layer consists of OFI (OpenFabrics Interfaces), UCX (Unified Communication X), and Portal network drivers. We support the OFI driver for the Tofu-D implementation. The "tofu" provider in OFI is our Tofu-D implementation. The almost same capabilities but much simpler user API is also provided, named "UTF."

1.3.1.1 OFI

OFI (OpenFabrics Interfaces) is a framework for high-performance communication, and Libfabric is its API. To make the libfabric interface available for a new network interface, we just implement the low-level implementation module, called "provider." The latest version at writing time is v1.10.1, but we have been using v1.8.0.

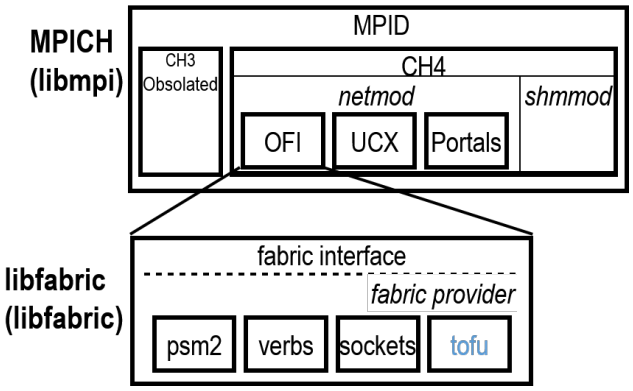


Figure 1.1: MPICH Software Architecture

The libfabric is capable of a point-to-point message passing function. A communication operation is divided into two phases: i) posting an operation, e.g., send, receive, and remote operation, and ii) receiving its completion. Those operations are non-blocking. There are four-type communication methods as follows:

1. fi_msg (Message Queue), The message queue is a simple point-to-point message passing interface. Messages are enqueued and handled in the FIFO order. An interesting feature is a multi-receive capability. If a receive operation is issued with the multi-receive option, multiple messages can be received in the single receive operation. Incoming messages are sequentially stored in the receive buffer, and completion events are generated for received messages. In the MPICH implementation, this multi-receive feature is employed to realize an active message function for describing the MPI remote memory access feature such as MPI window.

2. fi_tagged (Tagged Message Queue), The tagged message queue is the same concept of MPI communication. Each message has an integer value, called "tag." The receive/send operation may specify the message tag. Sent messages are matched with its tag and receiver's specified tag. MPICH uses this feature for the implemenation of basic MPI point to point message function.

3. fi_rma (Remote Memory Access), This is remote memory get and put features. MPI remote put and get operations can be implemented using this feature.

4. fi_atomic (Atomic), Atomic operations are operations on the remote memory regions, such as atomic-add and compare-and-swap. Those operations can realize MPI atomic operations, such as MPI_Fetch_and_op and MPI_Compare_and_swap.

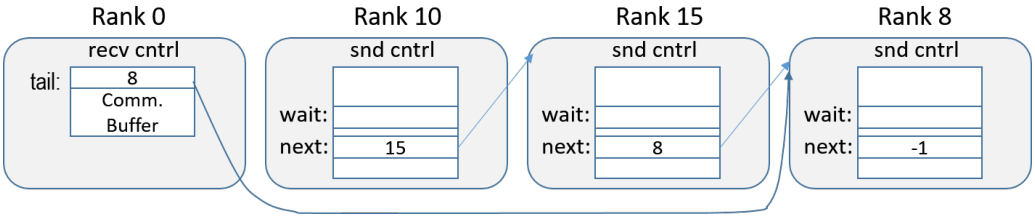


Figure 1.2: Chain Mode

1.3.1.2 uTofu and Tofu Provider

The "uTofu" library is the lowest user API for the Tofu-D interconnect interface. It provides the basic remote put and get operations, remote atomic operations, and barrier synchronization mechanisms. All memory regions, accessed by the Tofu-D interface, must be registered prior to the interface accessing. When a memory region is registered, the corresponding address defined by uTofu is associated with the registed virtual address. This address is called a "steering address."

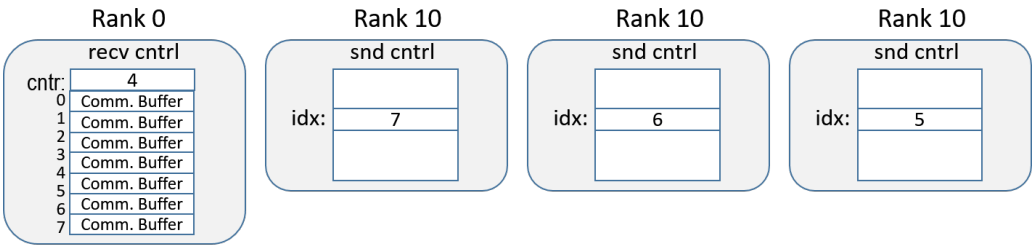


Figure 1.3: Reserve Mode

When a memory region is registered with a tag, an integer value, the steering address is identical over all other processes. We can create a common memory region over all processes using this feature without any extra processing. Unlike other interconnect interfaces, such as Infiniband, no message passing capability is provided in the Tofu-D interconnect. The fi_msg and fi_tagged operations are implemented in two ways as follows:

1. Chain mode
   In this mode, the single receive buffer is shared by all the processes. The waiting processes form a distributed wait queue using the uTofu atomic operation as shown in Figure 1.2. In this figure, rank 10 is sending messages to rank 0, and rank 15 is waiting for completion of rank 10's send operation, and rank 8 is the last process waiting for the other processes' completions.

2. Reserve mode
   Unlike the Chain mode, the limited common receive buffers are created to be accessed by other processes in the Reserve mode. Prior to send the first message to the receiver, the sender must reserve a receive buffer from the common receive buffers using the uTofu atomic operation. The reserved receive buffer is never freed. Figure 1.3 depicts the same situation of Figure 1.2. In this figure, the number of the common receive buffers is eight, and three of them have been reserved for ranks, 10, 15, and 8.

1.3.1.3 Implementation Status

The "tofu" driver and "UTF" have been implemented in FY2019. In FY2020, after evaluating those implementations, we will optimize those libraries.

1.3.2 Sharing Virtual Address Space

The two most common parallel execution models are multiprocess (MPI) and multithread (OpenMP). The multiprocess model allows each process to own a private address space, and the multithreaded model shares all address space by default. In the multiprocess model inter-process communication is inefficient because a process cannot access data owned by the other processes. In the multithread model threads share the same virtual address space and can access the all data which may incur lock contention overhead. Thus, both models have advantage and disadvantage. We propose a new implementation of the third model, called *Process-in-Process (PiP)* to take the best of two worlds, mutiprocess and multithread. In PiP, processes are mapped into a single virtual address space (VAS), but each process still owns its private storage. The idea of address-space sharing is not new. What makes PiP unique, however, is that its design is completely in user space, making it a portable and practical approach for large supercomputing systems.

In this fiscal year, we explore the potential of the PiP programming model. There are two research topics here, but both are close related with another. Unlike multithread model, PiP allows for two or more different programs to share the same virtual address space. Since they share the same virtual address space, a PiP task can context-switch to the another at user-level. This looks like another user-level thread (ULT) model and this is called user-level process (ULP). In ULT and ULP, some kernel threads schedules ULTs or ULPs. When a current task calls a systemcall to access the resource information resides in the OS kernel, it returns the information of the scheduling thread, not the one associated with the calling task. When a ULT or ULP migrates to the another kernel thread, then systemcalls return the information of the migrated kernel thread, not the information associated with the calling task. Thus there is no such consistency in the user-level implementations. In multithread model, regardless to its implementation of kernel-level or user-level, most kernel resources (e.g.,

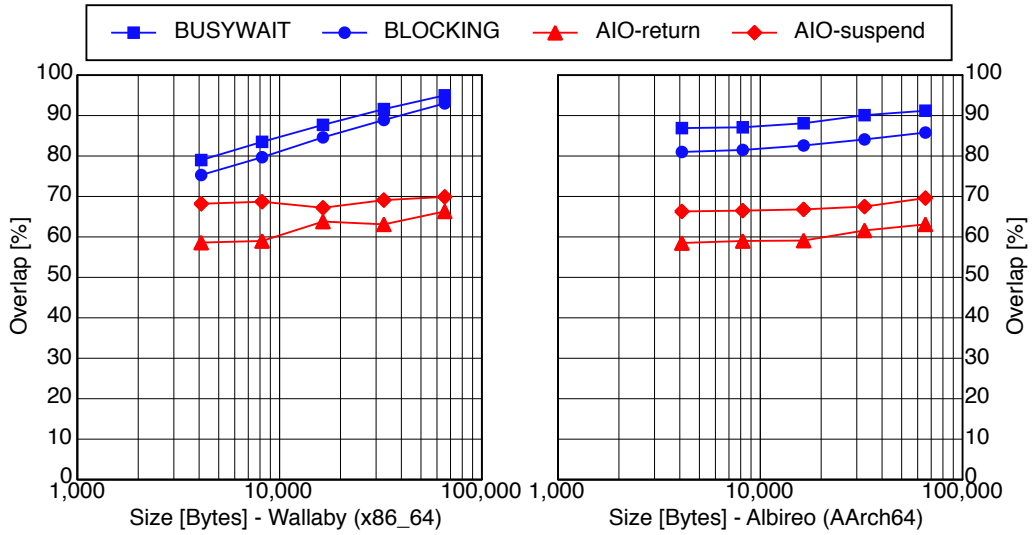


Figure 1.4: Comparing overlap ratio between Linux AIO and PiP/ULP (Wallaby: Intel Xeon E5-2650 v2, Albireo: AMD Opteron A1170 - Cortex-A57)

file descriptors) are shared and the systemcall consistency may not cause a severe problem. In multiprocess model, however, most of the kernel resources are not shared and the systemcall consistency can cause a big problem. To challenge this issue, PiP/ULP implements Bi-Level Thread/Task (BLT) where a user-level thread can become a kernel-level thread (KLT) and vice versa. When a ULP tries to call a systemcall, it becomes a KLT and the systemcall is called. When the systemcall returns it becomes a ULT again.

One of the well-known issues in ULT implementations is blocking systemcalls. In many cases, multithreads are oversubscribed to hide the sytemcall latency. When a ULT calls a blocking systemcall, the scheduling task is blocked and there is no chance for the other eligible-to-run ULTs to be scheduled and wasting CPU resources. The above mentioned BLT on PiP can also solve this issue in addition to the systemcall consistency issue.

Figure 1.4 shows the overlap ratios of Linux AIO calls (`aio_write` followed by `aio_return` or `aio_suspend`) and PiP (blocking or busy-waiting kernel thread). As shown, PiP implementation outperforms the Linux AIO cases.

1.3.2.1 CEA-Riken Collaboration

CEA has been developing MPC which has the same goal to implement shared address space as PiP does. PiP's approach is to implement a user-level library while MPC's approach is to implement another programming language based on the multithread model. It is very natural to collaborate with the other since both have the same goal.

As described above, PiP has succeeded to implement BLT and ULP on PiP. CEA is very interested in those idea and they are working to have the similar functionalities of PiP's BLT and ULP in their MPC implementation.

1.3.2.2 DOE-MEXT Collaboration

Balancing communication workload among MPI processes is always a critical point during the development of High-Performance Computing (HPC) applications in order to avoid resource waste and achieve the best performance. However, because of many factors such as complex network interconnections and irregularity of HPC applications, fully achieving communication workload balance in practice is nearly impossible. Although interprocess work stealing is a promising solution, current shared-memory techniques that lack necessary flexibility or cause inefficiency during data access cannot provide an applicable and efficient process-level work-stealing solution; in addition, existing thread-based work stealing methods cannot be efficiently applied to MPI-layer task model. To solve this problem, we propose a new interprocess work-stealing method using PiP to balance communication workload among processes on MPI layers.

Figure 1.5 shows the performance comparison between the MPICH enhanced with the proposed technique and existing AMPI on the Minighost micron benchmark program. As shown in this figure, the proposed

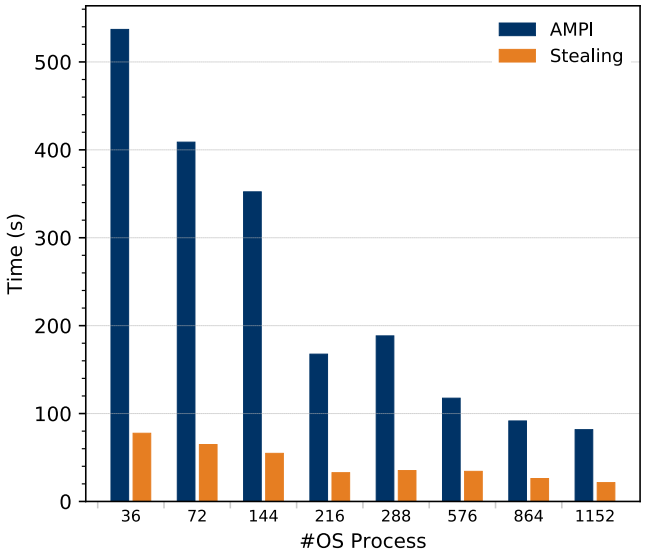


Figure 1.5: Performance comparison of AMPI and our method with Minighost strong scaling test. (Both use the same number of OS processes)

technique outperforms AMPI.

1.3.3 IHK/McKernel

This Section describes activities related to IHK/McKernel. Specifically, development items and results from a large-scale evaluation of a parallel multi-grid application on the Oakforest-PACS supercomputer are discussed.

1.3.3.1 Development Activities

Most of the fiscal year 2019 has been spent on porting McKernel to the ARM architecture using Marvell's ThunderX-2 system as well as a prototype cluster of the Fujitsu A64FX platform. Some of the major development items are as follows.

- **Memory semantics:** The ARM architecture differs from x86 in many ways. Most notably ARM has a more relaxed memory semantics than x86 which implies the need for careful and more explicit inter-core synchronization. For example, as opposed to x86, during bringup of CPU cores one needs to ensure that contents of the last level cache must be flushed to memory before any secondary CPU accesses the corresponding memory area. Another implication of the relaxed memory semantics is the requirement for explicit usage of memory barriers in situations where on x86 the order of memory operations would otherwise proceed sequentially.

- **Overlay filesystem:** The overlay filesystem support that was originally implemented as modification to the Linux kernel overlayfs module has been moved to user-space so that portability across different Linux kernel versions is improved.

- **Performance counters:** Support for the ARM performance counters has been added via an implementation of the `perf_event_open()` system call and surrounding infrastructure.

- **Utility thread interface:** An implementation for for the utility thread interface for offloading helper threads to Linux cores has been added for the ARM architecture.

- **Live kernel debugger:** An extension to the `eclair` kernel dump inspection tool has been implemented that enables live debugging of McKernel with support for resuming execution.

1.3.3.2 Parallel Multigrid Methods

Multigrid is a scalable method for solving linear equations and preconditioning Krylov iterative linear solvers, and is especially suitable for large-scale problems. The parallel multigrid method is expected to be one of the powerful tools on exa-scale systems. We previously proposed a new format for sparse matrix storage based on sliced ELL, which optimized the serial communication on shared memory systems, and hierarchical coarse grid aggregation (hCGA) has been introduced for optimization of parallel communication by message passing. The proposed methods are implemented for pGW3D-FVM that solves ground-water flow in a three-dimensional heterogeneous porous medium by the finite volume method, and by using parallel conjugate gradient (CG) solver with multigrid preconditioner (MGCG).

With hCGA, high parallel performance can be achieved even when the number of MPI processes is on the order of ten thousands by introducing two layers of parallel hierarchical levels. However, if the number of MPI processes is orders of magnitude higher, e.g., on future exa-scale systems, the computational overhead of the coarse grid solver can become significant. In this publication, we proposed the adaptive multilevel hCGA (AM-hCGA) method that introduces three or more layers of parallel hierarchical levels. We evaluated the new method on the Oakforest-PACS (OFP) system (JCAHPC) and examined the impact of the IHK/McKernel operating system.

Figure 1.6 summarizes our application level results. Each plate shows computation time for weak scaling of optimum cases for hCGA, and AM-hCGA, with and without IHK/Mckernel. Computation time for MGCG is normalized by the time of optimum case of hCGA without IHK/McKernel at each core number. Therefore, the values of Y-axis (Ratio) are always equal to 1.00 for hCGA without IHK/McKernel. Effects of IHK/McKernel for improvement of the performance of hCGA at 131,072 cores (2,048 nodes) is approximately 4% for Medium (m), 17% for Small (s), and 22% for Tiny (t) problem sizes.

1.3.4 Utility Thread Offloading Interface

The number of cores per node in HPC system has increased to the order of tens to hundreds. This abundance makes it possible to dedicate some cores to helper tasks to increase application performance. For example, MPI asynchronous progress threads can be put on those cores to parallelize a part of communication processing.

Putting those threads to appropriate cores is critical to the performance of this technique and needs the following additional cares. First, those threads should be identified as helper threads and treated in a special way so that they are put on the dedicated cores for the first place. Second, the thread-core mapping should be chosen to minimize the cost of the communication with those threads and the related components, e.g., the computation threads or the interconnect.

While there are the existing implementations, they are not done in an application transparent way, i.e., done by modifying application code or command line. In addition to that, they are not done in an easy way or in an across-systems-portable way because there is no simple and abstracted interface.

To solve those issues, we introduce an application-transparent and portable interface, called Utility Thread offloading Interface (UTI), and the library serving to runtime, e.g., MPI library. The library is given abstracted info by the runtime and performs the core allocation.

This is on-going work and the followings were done during this fiscal year.

1. Submit a paper about cross-layer resource coordinator which incorporating the UTI ideas. This is done with LLNL, Intel, CEA, INRIA people.
2. Improve prototype implementation of the library by getting the feedbacks from ANL people.

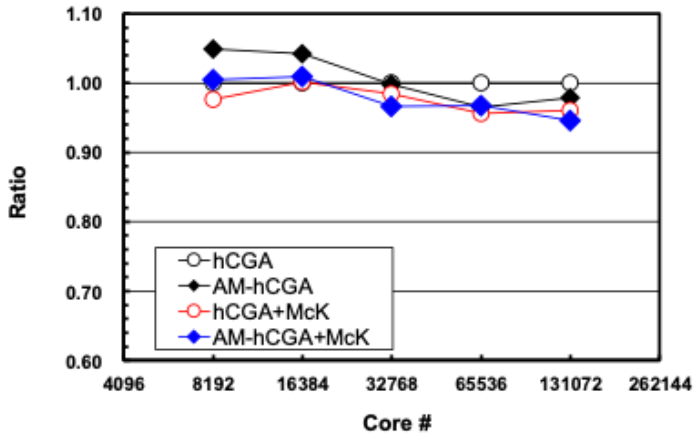
1.4 Schedule and Future Plan

We continue to enhance performance and functionality of our system software stack for Fugaku and other computer architectures, such as Intel Xeon.

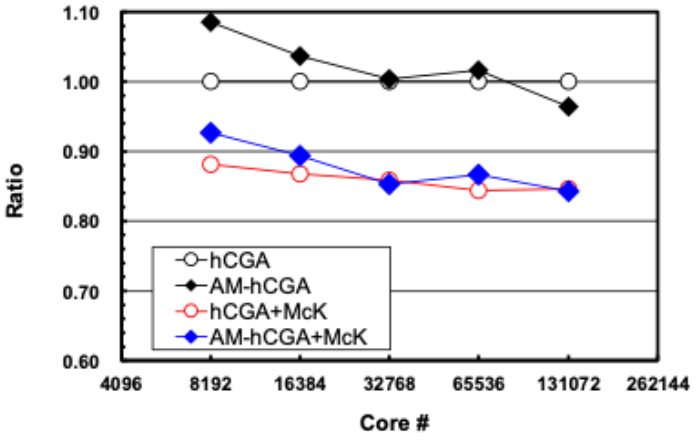
1.5 Publications

1.5.1 Articles/Journal

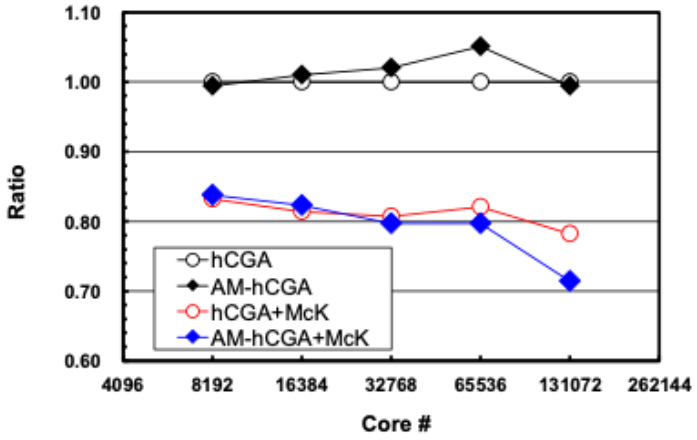
[1] A. Hori, K. Yoshinaga, T, Herault, A. Bouteiller, G. Bosilca, and Y. Ishikawa, Overhead of using spare nodes, International Journal of High Performance Computing Applications (IJHPCA), vol. 34(2) (2020).



(a) Medium



(b) Small



(c) Tiny

Figure 1.6: Computation time for MGCG solvers up to 2,048 nodes (131,072 cores) of OFP, normalized by computation time for hCGA at each core number, hCGA/AM-hCGA: without IHK/McKernel, hCGA/AM-hCGA+McK: with IHK/McKernel, (a) Medium (m), (b) Small (s), (c)Tiny (t)

1.5.2 Conference Papers

[2] A. Hori, G. Bosilca, E. Jeannot, T. Ogura, and Y. Ishikawa, Is Japanese HPC another Galapagos? - Interim Report of MPI International Survey -, IPSJ SIGHPC - SWoPP 2019, July, 2019.
[3] K. Nakajima, B. Gerofi, Y. Ishikawa and M. Horikoshi, "Parallel Multigrid Methods on Manycore Clusters with IHK/McKernel," 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), Denver, CO, USA, 2019, pp. 52-61, doi: 10.1109/ScalA49573.2019.00012.

1.5.3 Posters

1.5.4 Invited Talks

1.5.5 Oral Talks

[4] A. Hori, International Survey - MPI -, 9th JLESC Workshop, (Knoxville, USA, April, 2019) [5] A. Hori, A right decision is not always right, The All-RIKEN Workshop 2019 (Wako, Japan, Dec. 2019)

1.5.6 Software

`https://github.com/RIKEN-SysSoft` Distributing IHK/McKernel and PiP

1.5.7 Patents