Chapter 1

System Software Research Team

1.1 Members

Yutaka Ishikawa (Team Leader)
Masamichi Takagi (Senior Scientist)
Atsushi Hori (Research Scientist)
Balazs Gerofi (Research Scientist)
Yoshiyuki Morie (Research Scientist)
Masayuki Hatanaka (Research & Development Scientist)
Takahiro Ogura (Research & Development Scientist)
Tatiana Martsinkevich (Postdoctoral Researcher)

The system software research has been conducted in cooperation with the following members of the System Software Development Team in the Flagship 2020 project.

1.2 Research Activities

The system software 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 Persistent Collectives

The state-of-the-art interconnects, such as the Tofu2 interconnect of Fujitsu PRIMEHPC FX100, have not only an RDMA offloading capability but also support offloading collective operations. In addition to FIFO-based sequential execution for RDMA requests, such an interconnect supports complex scheduling for RDMA requests, including collective communications, by extending the FIFO-based scheduling to trigger and block the execution of the queued RDMA requests from remote nodes. It allows truly asynchronous progress of collective operations, which require issuing sequence of communication commands, without CPU assist.

The MPI forum, a community for standardizing MPI specification, has been discussing a new feature of collective operations, i.e., persistent collectives. For example, the persistent neighborhood all-to-all collective has the initialization function, MPI_Neighbor_alltoall_init(), that must be executed before initiating the collective operation. An application obtains a request object as a result of this initialization function, and it is used to
initiate the collective operation by MPI\_Start() function with that object. MPI\_Wait() function is used to wait for the completion. MPI\_Start() and MPI\_Wait() may be repeatedly invoked without re-initialization of the operation. This persistent collective operation was implemented using Tofu2 offloading capability in FY2017 [3]. The offloaded persistent broadcast, requiring more sophisticated algorithm to achieve low latency, was also implemented using Tofu2 offloading capability in FY2017 [6,7].

In order to integrate our past research results into the MPICH MPI implementation, we decided under DoE/MEXT collaboration that the existing Tofu access module is implemented as a “tofu” provider in Open Fabric Interface (OFI), libfabric, to combine with MPICH via OFI. OFI is one of the low-level communication interfaces supported by MPICH 3.3. In the rest of this subsection, an extended feature of OFI is described.

OFI has an API called triggered operations and Deferred Work Queue in which a communication operation is defined with a counter and its threshold. Figure 1.2 shows a definition of the deferred work queue entry. If the `triggering_cntr` in the `fi\_deferred\_work` structure reaches the `threshold`, the operation specified by `op\_type` and `op` is issued. This mechanism is implemented using the Tofu session mode CQ in which triggering counter is realized by the scheduling pointer of a CQ.

A persistent non-blocking operation can be realized using the Deferred Work Queue as follows. At the initialization function, such as MPI\_Neighbor\_alltoall\_init(), a series of triggered operations, implementing a collective operation, is created. This command sequences are registered using the `fi\_control()` libfabric interface at the MPI\_Start operation. MPI\_Wait operation waits for all completions of the commands. The triggered operations are handled by the Tofu network interface, and thus no CPU cycles are involved during the collective operation. However, the command sequences must be registered every execution of the MPI\_Start operation. We extended this API so that the registered commands are reused to eliminate re-registration overhead. A prototype implementation was conducted to proof this concept in FY 2018.

### 1.3.2 Data Transfer Framework

A common type of HPC applications is multi-component workflows in which each component performs a particular task, then passes the result of the computation to the next component for further processing. Components may be either loosely coupled, i.e., by using files to pass data, or they can use coupling software. An example of such applications is a severe weather prediction application called SCALE-LETKF that consists of two independently developed components. In each iteration SCALE performs ensemble simulations, outputs the
1.3. RESEARCH RESULTS AND ACHIEVEMENTS

results to a set of files, which LETKF reads and assimilates the result with observation data that arrives in real time at fixed intervals. LETKF then outputs the result to files that become the input for SCALE in the next iteration. The program, thus, has a strict timeliness requirement: one iteration, including all the I/O, needs to finish within a fixed time period - before next observation data arrives. This requirement becomes impossible to meet as the amount of I/O grows.

To solve this problem, we proposed the Data Transfer Framework (DTF) that silently bypasses file I/O by sending the data directly over network but does not require rewriting of the original I/O kernels [5]. Current operation of the DTF operates underneath the PnetCDF library - a popular I/O library used by SCALE-LETKF. The DTF intercepts PnetCDF’s I/O calls and sends the data to the processes in the other component using MPI library instead of performing file I/O. The DTF does require a slight modification to the user program. In particular, the user is required to add the following API calls to each component in the multi-component workflow:

1. DTF_INIT(config_file, component_name) - initializes the framework, should be called after MPI_Init();
2. DTF_FINALIZE() - finalizes the framework, should be called before MPI_Finalize();
3. DTF_TRANSFER(filename) - the data transfer is performed inside this call, should be called after corresponding PnetCDF read/write calls.

Additionally, to the aforementioned APIs the user needs to provide a DTF configuration file: a simple text file that describes the framework: it gives number and names of components, lists the files that create dependency between the two components and need to be transferred. The DTF is implemented as a library and it comes with a modified version of the PnetCDF library that allows intercepting of the I/O calls. We tested the performance of the DTF with SCALE-LETKF on the K computer. In the test case, the observation data with a resolution of 250 meter was assimilated. The tests were performed on three scales: with 60, 80 and 100 ensembles with 360 processes per ensemble. A total of 2.75 GB of data was moved between processes of one ensemble in one iteration. Figure 1.3 below demonstrates that even for the largest execution the time to transfer the data took less than 0.5 second.

1.3.3 Sharing Virtual Address Space

The two most common parallel execution models for CPUs today are multiprocess and multithread. The multiprocess model allows each process to own a private address space, and the multithreaded model shares all address space by default. We propose a new implementation of the third model, called Process-in-Process (PiP), where multiple processes are mapped into a single virtual address space (VAS). Thus, each process still owns its private storage but can directly access the storage of other processes in the same virtual address space. 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.

PiP allows multiple parallel tasks to execute the same or different programs in a shared virtual address space environment while maintaining privatized variable sets. The PiP task does not follow the definition of
conventional process or thread. For instance, PiP tasks share the same virtual address space whereas processes have isolated virtual address space; two PiP tasks can execute arbitrary programs in parallel in the same virtual address space, whereas threads must be derived from the same program. Moreover, in the thread mode, all variables are shared. In PiP, every PiP task has its own privatized variable set like the process model but all variables are also sharable since PiP tasks share the same VAS.

<table>
<thead>
<tr>
<th>Multi-Process</th>
<th>Virtual Address Space</th>
<th>Variables</th>
<th>Data</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>POSIX shmem</td>
<td>not shared</td>
<td>privatized</td>
<td>inaccessible</td>
<td>inaccessible only for newly allocated regions</td>
</tr>
<tr>
<td>XPMEM</td>
<td>shared</td>
<td>shared, accessible</td>
<td>accessible</td>
<td>Linux Kernel Module</td>
</tr>
<tr>
<td>Multi-Thread</td>
<td>(default)</td>
<td>shared, privatized, accessible</td>
<td>accessibile</td>
<td>Compilers needed</td>
</tr>
<tr>
<td>SMARTMAP PVAS</td>
<td>shared</td>
<td>privatized, accessible</td>
<td>accessible</td>
<td>Kitten OS patched Linux Kernel</td>
</tr>
<tr>
<td>Process-in-Process</td>
<td>shared</td>
<td>privatized, accessible</td>
<td>accessible</td>
<td>OS and language system independent</td>
</tr>
</tbody>
</table>

Table 1.1: Summary of current techniques and PiP

As listed in Table 1.1, two memory-mapping techniques have been widely used for data sharing in the multiprocess model. POSIX shmem includes System-V IPC and UNIX shared mmap. It is a general term to `mmap()` the memory pages owned by another process. This technique allows newly allocated memory segments (i.e., data) to be shared. However, a process cannot access statically allocated variables of the other processes. Moreover, the setup cost for the memory mapping is high. XPMEM is the other well-known approach. It allows processes to map arbitrarily memory regions (i.e., both data and variables) owned by other processes. These mappings involve costly system calls to the kernel module, however.

Two OS-based VAS-sharing techniques have been studied in the multiprocess model. SMARTMAP is a built-in function of the Kitten lightweight OS kernel that exploits the PT structure of the x86 architecture. PVAS provides functionality similar to that of SMARTMAP, but it is implemented as a patched Linux kernel. MPC is a thread-based language-processing system designed for hybrid MPI and OpenMP programming. It consists of custom compilers, linker, and runtime libraries; thus the maintenance is costly.

The PiP approach can be categorized into the process-based approach. It avoids the overhead of maintaining thread-safety and existing software stacks can be also used. PiP is implemented at user-level and requires neither the development of a new OS kernel, nor a patch to existing kernels. These properties make PiP highly practical.

1.3.3.1 DOE-MEXT Collaboration

SNAP\(^1\) is a proxy application that models the performance of modern discrete ordinates neutral particle transport application. The original implementation uses the hybrid MPI+OpenMP model where the spatial domain is partitioned across MPI ranks and traversed with sweeps along the angular domain. We updated the SNAP code to be PiP-aware. We compared the PiP-aware version (MPI+PiPs) with the conventional thread-based approach (MPI+OMP), and measured the MPI-only model as a reference.

Figure 1.4 reports the solve time of 10 time steps running on Oakforest-PACS (OFP). MPI+PiPs always outperforms MPI+OMP. Especially when scaling to a large number of cores, the communication becomes dominant, and thus PiP delivers more benefit, contributing close to 3.2x performance speedup on 256 cores (4 nodes).

The reduced speedup on 1,024 cores (i.e., 2.2x) indicates that the contention overhead of multithreaded MPI is optimized by the priority locks. Nevertheless, the network utilization issue still exists and relies on the support from PiP. We note that the MPI-only model shows faster solve time than that of the hybrid approaches on a single node (up to 64 cores). The reason is that it assigned every energy domain to a different process whereas the hybrid approaches divided each domain and distributed to four PiP tasks or threads. Thus, the former may

\(^1\) https://github.com/lanl/SNAP
1.3. RESEARCH RESULTS AND ACHIEVEMENTS

1.3.1 Strong Scaling of Hybrid MPI+OpenMP SNAP on OFP.

![Figure 1.4: Strong scaling of hybrid MPI+OpenMP SNAP on OFP.](image)

The figure shows the solve time (in seconds) for different numbers of cores using MPI-only, MPI+OMP, MPI+PiPs, and MPI+PiPs speedup. The speedup over MPI+OMP is also shown. The number of cores varies from 16 to 1024, each with a different configuration (e.g., 16(4x4x2x2), 64(8x8;4x4x4), 256(16x16;4x8x8), 1024(32x32;4x16x16)).

1.3.2 CEA-Riken Collaboration

As shown in Table 1.1, CEA is also developing MPC to provide the same address-space sharing technique, but with the different approach from PiP. CEA, however, has been developing MPC for years and has wider experiences with this model.

One big issue of address-space sharing model is that a malloc()ed region can be passed to the other task sharing the same address space, but the passed task is unable to free() it. This is because each task has a different GLIBC instance and the malloc information is not shared among tasks. In PiP, this issue is solved in a very naive way.

If this restriction could be lifted, then users would have more programming flexibility with this execution model. MPC development team already noticed and solved this problem. They have developed another malloc library, named MPC Allocator. The benefit of having MPC Allocator is not only solving the issue but also the better performance of MPC Allocator than that of GLIBC.

In this collaboration scheme, MPC Allocator was firstly decoupled from the MPC runtime library as a independent library. This was done by CEA, and then Riken checked if MPC Allocator can work on PiP. Eventually we found that MPC Allocator can work on PiP. Figure 1.5 shows the performance difference between the PiP malloc()/free() and MPC Allocator running on PiP. As shown in this figure, MPC Allocator performs much better than that of PiP’s naive implementation (PiP_malloc and PiP_free).
1.3.4 IHK/McKernel

This Section describes recent research results related to IHK/McKernel. Specifically, results of a custom precise event based sampling (PEBS) device driver for scalable, real-time memory access tracking are presented in Section 1.3.4.1 published in [6].

1.3.4.1 Scalable Memory Access Tracking

The past decade has brought an explosion of new memory technologies. Various high-bandwidth memory types, e.g., 3D stacked DRAM (HBM), GDDR and multi-channel DRAM (MCDRAM) as well as byte addressable non-volatile storage class memories (SCM), e.g., phase-change memory (PCM), resistive RAM (ReRAM) and the recent 3D XPoint, are already in production or expected to become available in the near future.

Management of such heterogeneous memory types is a major challenge for application developers, not only in terms of placing data structures into the most suitable memory but also to adaptively move content as application characteristics change in time. Operating system and/or runtime level solutions that optimize memory allocations and data movement by transparently mapping application behavior to the underlying hardware are thus highly desired.

One of the basic requirements of a system level solution is the ability to track the application’s memory access patterns in real-time with low overhead. Intel’s Processor Event-Based Sampling (PEBS) is an extension to hardware performance counters that enables sampling the internal execution state of the CPU (including the most recent virtual address accessed) and periodically storing a snapshot of it into main memory. The overhead of PEBS has been the focus of previous works, however, not in the context of large-scale high-performance computing (HPC).

We have implemented a custom PEBS driver in the IHK/McKernel lightweight multi-kernel operating system and performed systematic evaluation of PEBS’ overhead on a number of real HPC applications running at large scale. We demonstration of captured memory access patterns as the function of different PEBS parameters.

Figure 1.6 summarizes our application level findings. The X-axis represents node counts while the Y-axis shows relative overhead compared to the baseline performance. For each bar in the plot, the legend indicates the PEBS reset value and the PEBS buffer size used in the given experiment. The general tendency of overhead for most of the measurements matched our expectations, i.e., the most influential factor in performance overhead is the PEBS reset value, whose impact can be relaxed to some extent by adjusting the PEBS buffer size.

Across all workloads, we observe the largest overhead on GeoFEM (shown in Figure 1.6a) when running with the lowest PEBS reset value of 64 and the smallest PEBS buffer of 8kB, where the overhead peaked at 10.2%. Nevertheless, even for GeoFEM a less aggressive PEBS configuration, e.g., a reset value of 256 with 32kB PEBS buffer size induces only up to 4% overhead.

We concluded that PEBS efficiency matches the basic requirements to be feasible for heterogeneous memory management but further work is necessary to quantify the additional overhead associated with using the recorded data at runtime.

1.3.5 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 spare some cores for helper threads to increase application performance. A typical example is MPI asynchronous progress thread which performs communication processing in parallel with computation.

Allocating appropriate core is crucial to the technique for two reasons. The first one is that those threads should be put on the dedicated cores for the first place so as not to interfere with computation threads. The second one is that those threads need to communicate computation threads or interconnect and therefore they should be put to cores near the communication target to minimize communication overhead.

However, the implementation is not application transparent because it’s done typically by modifying application code or command line. And it is not easy nor portable across systems because no simple, abstracted interface is available.

To respond to this situation, we introduce a application-transparent and portable interface, called Utility Thread Offloading Interface (UTI), and the corresponding library. The library is given abstracted info by runtime (e.g. MPI library) and performs core allocation.

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

1. Review UTI with CEA. How to incorporate the capability to dynamically change the boundary between helper and computation cores is discussed.
2. Improve prototype implementation of the library with LLNL. Some issues are reported by the LLNL researcher.

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


