Memory-centric Architecture for Disaggregated Computers

Teruaki Ishizaki and Yoshiro Yamabe

Abstract

At first glance, most of the technical components of a disaggregated computer seem to be hardware. However, simply using current software cannot enable efficient computing even if each hardware resource is connected directly with optical fibers. This article introduces the problems of current software that has evolved on the premise of high-speed central processing units and describes memory-centric architecture as a new data-exchange model for a disaggregated computer.

Keywords: disaggregated computer, memory-centric architecture, shared memory

1. Memory-centric architecture

Central processing unit (CPU) performance has evolved rapidly, as Moore's Law states that semiconductor integration will double every 18 months. Compared with this CPU evolution, memory and storage networks have evolved slowly. Therefore, most current software is designed on the premise of high-performance CPUs and other low-speed devices and the policy of shortening the processing time of other devices by conducting as many calculations on the CPU as possible.

When executing storage input/output (I/O), for example, the CPU efficiently adjusts the unit size of the I/O and request amount for the I/O target data stored in the memory. For processing that requires multiple accelerators, the software on the CPU controls the accelerators. Such a processing model, in which the CPU intervenes in processing, is called a CPU-centric computing model (**Fig. 1(a)**).

As a general software design, the CPU is designed to mediate processing. However, the growth of CPU core performance has slowed, which is said to be the limit of Moore's Law. Non-volatile memory, which is a high-speed storage, and accelerators, such as field-programmable gate arrays (FPGAs) and graphics processing units (GPUs), are rapidly evolving. A new software-processing model for improving the performance of various accelerators is also becoming more important.

Therefore, we are investigating a memory-centric architecture that enables efficient cooperation between various accelerators via the main memory. This architecture focuses on the main memory where an accelerator starts processing, enabling efficient data exchange via the main memory when accelerators are linked.

Memory-centric architecture is a data-exchange model in which the sender accelerator inputs data into the main memory (shared memory) and the receiver accelerator autonomously acquires and calculates the data. Taking the example of receiving data from another node with a network interface card (NIC) and processing it with an FPGA, in the CPU-centric computing model, the CPU controls the reception of the NIC and the execution of processing to the FPGA. However, memory-centric architecture requires only two minimum processes, NIC network-reception processing and FPGA-arithmetic processing, so CPU resource consumption and processing-time reduction to mediate the processing can be expected (**Fig. 1(b)**).

When multiple accelerators execute parallel processing using the data input to the memory, the sender accelerator only needs to be placed once in the shared memory area, which is a more efficient processing model (Fig. 1(b)).

A concept close to memory-centric architecture is Hewlett Packard Enterprise (HPE)'s Memory Driven

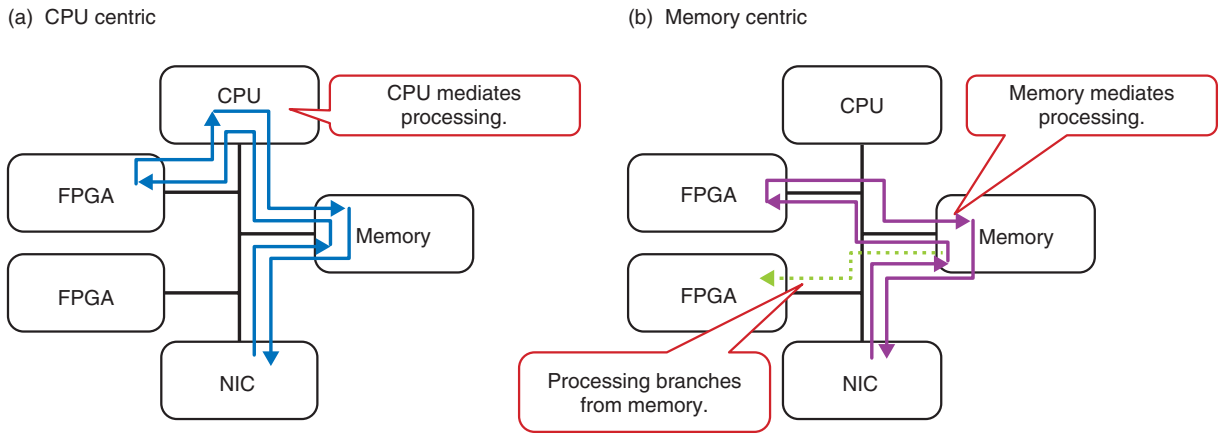


Fig. 1. Comparison of processing flow.

Computing (MDC) [1]. MDC is a concept model based on hardware architecture, such as a configuration in which a large memory pool is placed in the center of all processors. Memory-centric architecture, on the other hand, focuses on data control by software for exchanging data between accelerators via memory.

2. Evaluation on the effectiveness of memory-centric architecture

We evaluated the effectiveness of memory-centric architecture by using Sparkle [2, 3], which is an extension of Apache Spark (open source software for distributed data processing), as current software that is close to the concept of efficient data exchange via shared memory.

Apache Spark consists of a map phase that executes data processing in each worker process, shuffle phase that distributes the processing results executed in the worker process to the required processes, and reduce phase that executes operations on the basis of the results collected from each worker process. The shuffle phase is a process for exchanging a large amount of data by network communication using the TCP/IP (Transmission Control Protocol/Internet Protocol) between worker processes distributed over multiple nodes. Therefore, the shuffle phase's cost is very high in terms of CPU-processing delay related to network communication and CPU resource consumption. Sparkle is an extension that enables shuffle processing via shared memory for solving this problem.

Sparkle is software developed by HPE for MDC and is open source. However, its further development was suspended in 2016 when our evaluation study started, so we had to make some bug fixes to evaluate it [4].

Figure 2 is a comparison of Apache Spark (Fig. 2(a)) and Sparkle (Fig. 2(b)) data-exchange models assuming that multiple worker processes are running on the same server. Since connections are created between processes that exchange data, a large number of connections will be created due to the increase in worker processes as the processing scale increases. The process on the receiving side can refer to the data by receiving data through the network. However, to carry out network processing, it is necessary to execute data operations for transfer (protocol stack processing, memory copy to kernel space, etc.) on the CPU on both the transmitting and receiving sides. Therefore, this process increases delay. Since Sparkle can be referenced by other processes by arranging the data on the shared memory by the sending process, it is possible for the receiving process to acquire the data by referring to the memory area required. In other words, each process simply places the calculation result in the shared memory, and all processes can refer to the data without the CPU executing data operations for data transfer only, enabling efficient transmission and reception.

Graphs of Apache Spark and Sparkle performance test results are shown in **Figs. 3** and **4**. Figure 3 shows the performance comparison results of five basic processes including shuffle processing. The range of performance improvement differed since the ratio of shuffle processing to the entire processing differed depending on the processing. In particular, Sparkle

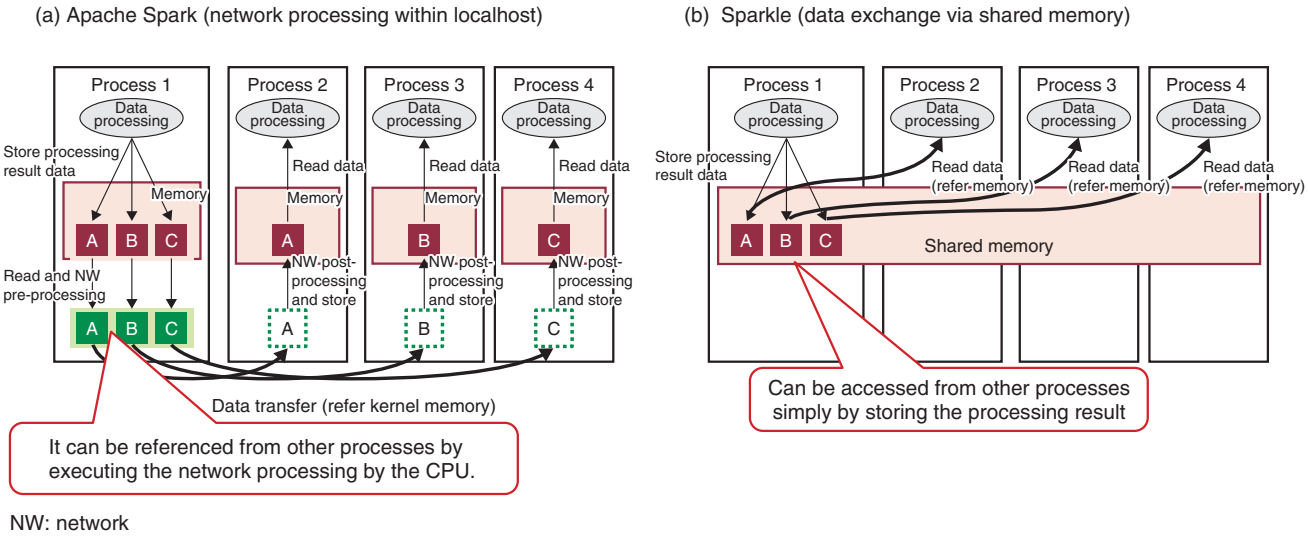


Fig. 2. Difference in data-exchange models.

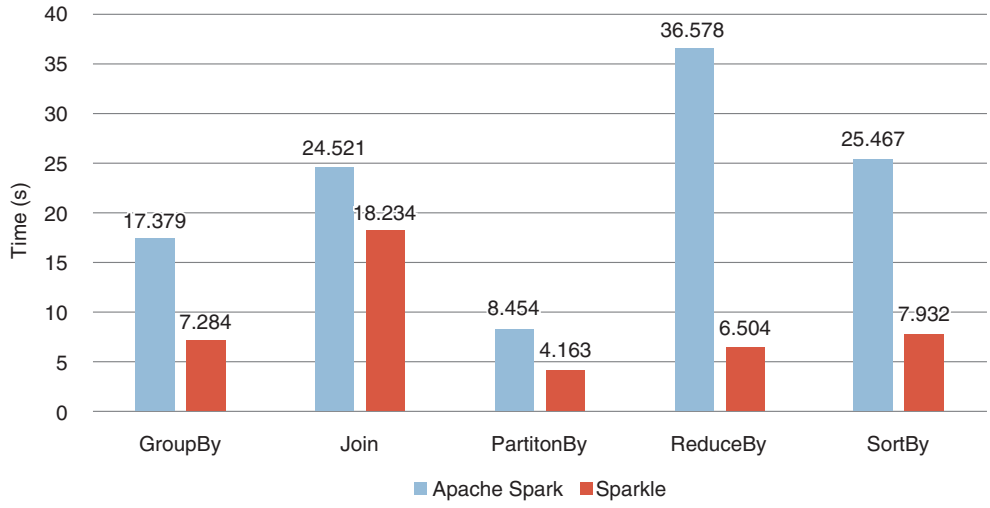


Fig. 3. Comparison of micro-benchmark result.

was about 6 times faster than Apache Spark in the case of ReduceBy, resulting in a significant performance improvement. Figure 4 shows the results of comparing the streaming processing performance using Spark Streaming. In this measurement, Sparkle had about twice the performance improvement compared with Apache Spark, and is expected to be effective for processing with strict delay requirements.

It was reconfirmed that the construction of a data-exchange model using shared memory enables efficient data exchange and reduction in delay, but there are issues in its construction. One of the biggest issues is the management of shared memory. It is common for the same file to be mapped by multiple processes when using shared memory, and it is necessary to use general C language pointer access to access the shared memory area. Although this can be described in general C language notation, memory management, such as access control of the shared memory area between multiple processes, free area management, and allocation, is not supported. Therefore, each application programmer needs to implement

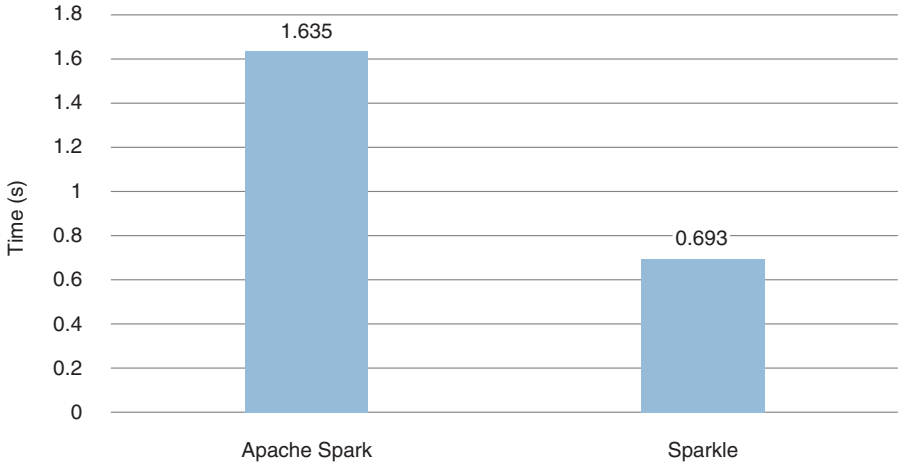


Fig. 4. Comparison of macro-benchmark using Spark Streaming.

their own memory management function to prevent data overwriting and double allocation of the memory area. The shared memory can be also used within a single server. To execute distributed processing across multiple servers, it is necessary to expand the function to enable such processing.

Research on software that does not impair the low latency of shared memory and that programmers can easily benefit from has become an important research subject for achieving memory centricity.

3. Conclusion

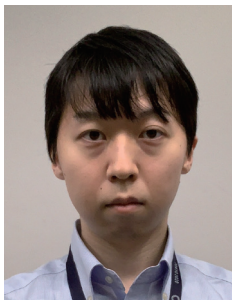
From the evaluation results of Sparkle, the effectiveness of using a data-exchange model via shared memory for a CPU was confirmed. As a study on memory-centric architecture, we will consider expanding this data-exchange model via shared memory to other accelerators such as FPGAs and study as a core technology that connects various hardware processes. Memory-centric architecture is positioned as an elemental technology required for disaggregated computers that support IOWN (Innovative Optical and Wireless Network) and is positioned as a software technology for using computer resources independently rather than on a server-chassis basis. By connecting and using accelerators without using a CPU through software control, we aim to reduce delay and CPU consumption.

References

[1] K. Keeton, "Memory-driven Computing," FAST 2017, Santa Clara, USA, Feb.–Mar. 2017.
[2] M. Kim, J. Li, H. Volos, M. Marwah, A. Ulanov, K. Keeton, J. Tucek, L. Cherkasova, L. Xu, and P. Fernand, "Sparkle: Optimizing Spark for Large Memory Machines and Analytics," arXiv preprint, arXiv:1708.05746, 2017.
[3] Hewlett Packard/Sparkle on GitHub, https://github.com/HewlettPackard/sparkle
[4] Sparkle-plugin on GitHub, https://github.com/sparkle-plugin/sparkle

Teruaki Ishizaki
Senior Research Engineer, Distributed Computing Technology Project, NTT Software Innovation Center.

He received a B.E. and M.E. in mechanical and environmental informatics from Tokyo Institute of Technology in 2002 and 2004. He joined NTT Cyber Space Laboratories in 2004 and studied the Linux Kernel and virtual machine monitor. From 2010 to 2013, he joined the cloud service division at NTT Communications and developed and maintained cloud and distributed storage services. He is currently studying a persistent memory programming model, remote direct memory access (RDMA) programming model, cloud-native computing, and memory-centric computing.



Yoshiro Yamabe
Research Engineer, Distributed Computing Technology Project, NTT Software Innovation Center.

He received a B.E. and M.E. in information and communication engineering from the University of Tokyo in 2014 and 2016. He joined NTT Software Innovation Center in 2016 and has studied an RDMA programming model, shared memory programming model, and memory-centric computing.