applied sciences

MDPI

Article

Empirical Performance Analysis of Collective Communication for Distributed Deep Learning in a Many-Core CPU Environment

Junghoon Woo, Hyeonseong Choi and Jaehwan Lee *

School of Electronics and Information Engineering, Korea Aerospace University, 76, Hanggongdaehak-ro, Deogyang-gu, Gyeonggi-do, Goyang-si 10540, Korea; starrystyle88@gmail.com (J.W.); chyon794@gmail.com (H.C.)

* Correspondence: jlee@kau.ac.kr; Tel.: +82-2-300-0122

Received: 28 August 2020; Accepted: 21 September 2020; Published: 25 September 2020

Abstract: To accommodate lots of training data and complex training models, "distributed" deep learning training has become employed more and more frequently. However, communication bottlenecks between distributed systems lead to poor performance of distributed deep learning training. In this study, we proposed a new collective communication method in a Python environment by utilizing Multi-Channel Dynamic Random Access Memory (MCDRAM) in Intel Xeon Phi Knights Landing processors. Major deep learning software platforms, such as TensorFlow and PyTorch, offer Python as a main development language, so we developed an efficient communication library by adapting Memkind library, which is a C-based library to utilize high-performance memory MCDRAM. For performance evaluation, we tested the popular collective communication methods in distributed deep learning, such as Broadcast, Gather, and AllReduce. We conducted experiments to analyze the effect of high-performance memory and processor location on communication performance. In addition, we analyze performance in a Docker environment for further relevance given the recent major trend of Cloud computing. By extensive experiments in our testbed, we confirmed that the communication in our proposed method showed performance improvement by up to 487%.

Keywords: Intel knights landing; distributed deep learning; python; collective communication; high bandwidth memory; docker; MPI

1. Introduction

The performance of deep learning [1] has improved significantly, thanks to the advanced neural network architecture, computer hardware (HW) performance improvement, and utilization of large-scale datasets. To accommodate massive amounts of training data on large-scale neural network models, most deep learning platforms, such as TensorFlow [2] and PyTorch [3], offer "distributed" training methods to exploit multiple cores and machines. When executing distributed deep learning, the computation of gradients is executed by hundreds of cores in worker nodes, and the computed gradients should be shared with other nodes to keep the updated gradients. Since updating fresh gradients across nodes should be done periodically, the collective communication performance can be a major performance bottleneck point in the distributed deep learning process [4]. Li et al. show a bottleneck of communication when training a deep learning model (AlexNet, ResNet-152 and VGG-16) through distributed deep learning in five machine cluster. In this training, about 75% of the total running time was communication time [4]. Therefore, efficient distributed deep learning requires to reduce communication time by optimizing communication layers.

Data parallelization is a representative distributed deep learning method. Data parallelization can be divided into a parameter server architecture and an AllReduce architecture according to the

communication method. In the parameter server architecture, there are parameter server processes and worker processes. The parameter server manages model parameters, and workers compute gradients. In deep learning systems, parameters are variables of the model to be trained. Gradients are values to find the optimal parameters obtained after back-propagation. Therefore, to train the deep learning model in the parameter server architecture, communication between the parameter server and the worker is required. In contrast, the AllReduce architecture does not have a parameter server to manage model parameters. The gradients computed by each worker are shared with each other through peer-to-peer (P-to-P) communication. Therefore, when performing distributed deep learning, various types of communication are performed, and an efficient communication method is required to improve the performance of the distributed deep learning system.

Python is a library-rich and object-oriented language available on a variety of platforms [5]. It is used in various fields along with computer science because it is easy to learn and use, especially for deep learning applications. TensorFlow and PyTorch, which are typical frameworks for deep learning, are also based on Python. For these reasons, Python should use multiple cores and simultaneous communications to support distributed deep learning efficiently. In Python-based distributed deep learning environments, we can use multi-processing or multi-threading methods. The major difference between the two methods is the way to share memory; multi-threading can share memory more efficiently compared to multi-processing. However, in Python environments, multi-threading has the disadvantage of limiting the use of computing resources because of Python's unique feature of Global Interpreter Lock (GIL) [6,7]. These characteristics make Python easier to use but also present obstacles to performance. Therefore, simply adopting multi-threading communications is not efficient in distributed deep learning training workloads.

In this paper, we propose a novel method of efficiently exchanging data when performing distributed deep learning in Python and many-core CPU environments, and we compare and analyze the proposed method through extensive experiments in various environments.

The contributions of this paper are as follows:

- First, we propose a new method of exchanging distributed deep learning data using Message Passing Interface (MPI) [8,9] that performs multi-processing instead of multi-threading, which has a problem with the GIL feature in Python. We developed Application Programming Interfaces (APIs) of Broadcast, Gather, and AllReduce collective communication [10], which are frequently used as MPI data communication methods in distributed deep learning. In addition, we compare and analyze Python's two MPI communication methods, buffer communication, and object communication.
- Second, we improve our Python-based MPI library by leveraging the advanced performance of On-package memory Multi-Channel Dynamic Random Access Memory (MCDRAM) of the Intel Xeon Phi Knights Landing processor [11], which is a typical many-core CPU used in supercomputing centers. To utilize MCDRAM, we adopted the Memkind library, a C-based library for memory allocation [12]. To use the Memkind library in a Python environment, Cython [13,14] is used to implement wrapping in Python.
- Third, for comparative analysis in various environments, we compare and analyze MPI communication performance by extensive experiments according to the Cluster mode and Memory mode of the Intel Xeon Phi Knights Landing processor. In addition, considering the practical distributed deep learning environments, we also conducted the experiment on the open source virtualization platform Docker [15] to see the performance difference in a Cloud computing environment.

The remainder of this paper is organized as follows. Section 2 presents related works. Section 3 explains the background of our study. Section 4 explains how we developed a Python-based MPI library to exploit MCDRAM and how we bound it with Docker Container. Section 5 describes the

experiment setup and compares the results to evaluate the proposed method with existing methods. Section 6 summarizes the experiment results and discussions. Section 7 concludes this paper.

2. Related Work

There is a study to improve and evaluate the communication performance between nodes in distributed deep learning. Ahn et al. [16] propose a new virtual shared memory framework called Soft Memory Box (SMB) that can share the memory of remote nodes between distributed processes in order to reduce the communication overhead of large-scale distributed deep learning. SMB proposed a new communication technology other than MPI by constructing separate servers and virtual devices. Our research and SMB have the same goal of improving data communication performance when deep learning in a distributed environment. However, there is a difference in that we proposed performance improvement based on the utilization of resources of high-performance computing and commonly used Python and MPI communication technologies.

There are studies to analyze and evaluate the performance of High-Bandwidth Memory (HBM). Peng et al. [17] compare the performance of HBM and DRAM and analyze the impact of HBM on High Performance Computing (HPC) applications. The results of this study show that applications using HBM achieve up to three times the performance of applications using only DRAM. We not only compare HBM and DRAM but also compare and analyze the performance of two memories in a Docker Container environment.

There is a study to evaluate communication performance in many-core CPU and high bandwidth memory environment. Cho et al. [18] compare the MPI communication performance according to the MVAPICH2 shared memory technology and the location of the many-core CPU processor. In the experiment, intra-node communication and inter-node communication are compared. In addition, this study evaluates experiments according to the location of the memory and processor. Unlike this study, we propose to improve the MPI performance in the Python environment, and we experiment and analyze according to the Cluster mode and Memory mode.

There is a study to evaluate the performance of the virtualization solution Container and Hypervisor. Awan et al. [19] compare the performance overhead of virtualization in Container-based and Hypervisor-based. The results of this study show that the average performance of Containers is superior to that of virtual machines. Therefore, in our study, we evaluate the performance in the Docker Container environment.

There are several studies evaluating communication performance using Containers in HPC environment. Zhang et al. [20] evaluate the performance of MPI in Singularity-based Containers. Experiments show that Singularity Container technology does not degrade MPI communication performance when using different Memory Modes (Flat, Cache) on Intel Xeon and Intel Xeon Knights Landing (KNL) platforms. However, our study evaluates and analyzes MPI performance in Docker Container. In addition, Zhang et al. [21] propose communication through shared memory and Cross Memory Attach (CMA) channels when executing MPI tasks in a multi-container environment. According to this study, communication performance is improved in a large-scale Docker Container environment. Our study compares and analyzes in multiple Docker Containers using Python environment and high-performance memory.

Li et al. propose In-Network Computing to Exchange and Process Training Information Of Neural Networks (INCEPTIONN) to optimize communication in distributed deep learning [4]. They propose a compression algorithm to optimize communication and implement the algorithm through hardware. In addition, through experiments, they show that INCEPTIONN can reduce communication time by up to 80.7% and improve speed by up to 3.1 times compared to conventional training systems. Unlike INCEPTIONN, in our study, we propose a software-based method to optimize communication in distributed deep learning by utilizing KNL's HBM and MPI.

3. Background

In this section, we describe distributed deep learning, Python GIL, MPI, and Intel Knights Landing.

3.1. Distributed Deep Learning

To achieve high accuracy using deep learning, the size of datasets and training models is increasing. For example, the ImageNet dataset, one of the most used datasets, consists of over 1.2 million images [22]. In addition, the amount of uploaded data from Social Networking Service (SNS) users is increasing day by day [23]. In the case of deep learning model, the AmoebaNet model, which shows the highest accuracy in recent years, consists of more than 550 M parameters [24]. Therefore, the computing power and memory resources required to perform deep learning are increasing [25,26]. In order to solve the increase in required resources, distributed deep learning using multiple workers is performed.

The purpose of a deep learning system is to find the value of the parameters that shows the highest accuracy. In deep learning, the parameter is the weight of the features of the input data. Gradients are the derivatives of the loss function based on model parameters. Therefore, training is the process to find a parameter that can minimize the loss function. Therefore, it is necessary to minimize the loss by applying gradients computed by distributed workers to the model parameters.

The distributed deep learning methods can be two-fold: data parallelization and model parallelization. In the case of data parallelization, the deep learning model is replicated across multiple workers [27,28]. Each worker trains a deep learning model with different input data. In the case of model parallelization, one training model is divided into multiple workers. Each worker trains part of a deep learning model with the same input data.

In data parallelization, workers compute gradients using different input data. Therefore, it is necessary to aggregate the gradients computed by each worker and apply the aggregated gradients to the model parameters of all workers. Data parallelization architecture can be divided according to the way gradients aggregation is performed. Typical data parallelization architectures are parameter servers and AllReduce [29,30]. In the parameter server architecture, processes are divided into two types—parameter servers and workers. The parameter server updates model parameters using gradients computed by workers and send the latest model parameters to workers. The worker computes gradients to update the model parameters using the input data and model parameters received from the parameter server, and after that, sends gradients to the parameter server. The parameter server architecture can operate both synchronously and asynchronously. In a synchronous way, the parameter server broadcasts model parameters to all workers and gathers computed gradients. In an asynchronous way, the parameter server exchanges model parameters and gradients with each worker through peer-to-peer communication.

In the case of the AllReduce architecture, there is no server that collects gradients computed by each worker and manages model parameters. In the AllReduce architecture, each worker shares gradients computed by each other through AllReduce based on P-to-P communication. Therefore, overhead due to communication between parameter server and worker can be reduced. However, the AllReduce architecture has a limitation that AllReduce architecture should operate only in a synchronous way.

As described above, distributed deep learning can be performed through various communication methods. Broadcast is a One-to-All communication method. In the synchronous parameter server architecture, the parameter server broadcasts the latest model parameters to all workers. Gather is an All-to-One communication. Gather is used when the parameter server collects the gradients computed by workers. AllReduce is an All-to-All communication. In the AllReduce architecture, AllReduce communication is used to share the gradients computed by each worker with other workers. In addition, as the scale of the distributed deep learning system increases, the amount of communication increases. Therefore, in order to mitigate the performance degradation due to communication when performing distributed deep learning, an efficient communication method must be proposed.

3.2. Python GIL

Python's Global Interpreter Lock (GIL) is a mutex that restricts access to Python objects to only one thread when multiple threads are used to execute Python code [6]. GIL is needed because CPython's memory management method is not thread-safe. CPython is the very first version of Python, which was originally written in C [31]. CPython is the standard implementation of Python to have basic functionality. In the early days of CPython's development, there was no concept of threads. Later, the thread concept was used by many developers as an easy and concise CPython language. Over time, there was a problem due to the concept of a thread, and GIL was created as a prompt solution to the problem. CPython needs to request access to the GIL to run multiple threads in one interpreter. When using asynchronous and multi-threaded code, Python developers can avoid process crashes by locking or deadlocking variables.

For this reason, GIL allows CPython to run only one thread when using multi-thread. It is not a problem when implementing a single thread using Python, but there is a problem of wasting resources when performing multi-thread in many-core CPU environments. There are various methods to solve this problem. As a first method, when performing multi-threading using a CPU, a task that requires a lot of computation is executed with C code out of GIL. The second method is to use multi-process instead of multi-thread. Since CPython 2.6, the multi-processing module has been added as a standard library. Multi-processing can share variables through queues or pipes, and it also has a lock object that locks the objects of the master process for use in other processes. However, multi-process has one important disadvantage; it has significant overhead in memory usage and execution time. Therefore, in many-core environments, the multi-processed Python execution is not an effective solution, especially when lots of data in memory should be shared with multi-processes.

3.3. MPI

Message Passing Interface (MPI) is a library for message passing and refers to the basic functions, Application Programming Interface (API), required for data exchange in distributed and parallel processing [9,10]. This standard supports both point-to-point and collective communication. The MPI application consists of several processes that exchange data with each other. Each process starts in parallel simultaneously when the MPI application is running and exchanges data with each other during execution. The advantage of MPI is that it can exchange data and information easily between multiple machines. Parallel MPI is used not only in parallel computers that exchange information at high speed, such as shared memory or InfiniBand, but also in PC clusters. MPI has various open libraries on different HW environments, such as MPICH, OpenMPI, and Intel MPI [32,33].

MPI allocates tasks based on process. Each process corresponds to a set called the communicator. A communicator defines a group of processes that have the ability to communicate. In the same communicator, messages can be exchanged with different processes. In the communicator, processes are identified by rank. There are two types of communication methods in MPI: point-to-point communication and collective communication. MPI_Send and MPI_Recv are common point-to-point communication methods of MPI. Group communication is communication in which all processes, which are communicators, participate together. MPI_Bcast, MPI_Gather, MPI_Allgather, MPI_Reduce, and MPI_AllReduce are all representative collective communications. Figure 1 shows the procedure for collective communication. The experiment presented in this study performs MPI_Bcast, MPI_Gather, and MPI_AllReduce communication. MPI_Bcast is a Broadcast communication that sends data of the root process to all other processes in the same communicator. MPI_Gather gathers data from multiple processes in the Communicator to the root process. MPI_AllReduce combines the values of all processes in the communicator and redistributes the results to all processes.

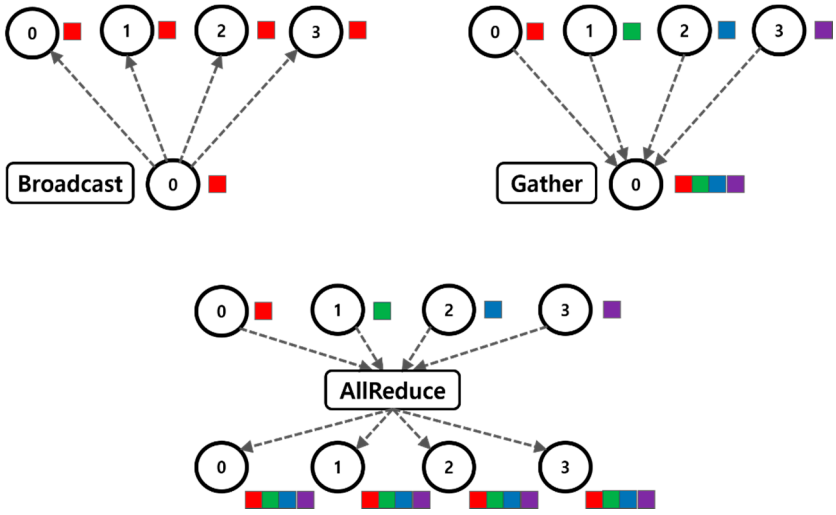


Figure 1. Message Passing Interface (MPI) collective communication.

3.4. Many-Core CPU: Knights Landing

The Intel Xeon Phi Knights Landing (KNL) processor is a many-core processor developed by Intel for use in supercomputers, high performance servers and advanced workstations [11]. Figure 2 shows the KNL diagram. A socket of KNL can have up to 36 active tiles, and each tile consists of two cores [34–36]. Thus, a Knights Landing socket can have up to 72 cores. The two cores in each tile communicate through the 2D Mesh Interconnect Architecture based on the Ring Architecture. The communication mesh is composed of four parallel networks. Each network provides different types of packet information, such as commands, data, and responses. It is optimized for Knight Landing traffic flows and protocols.

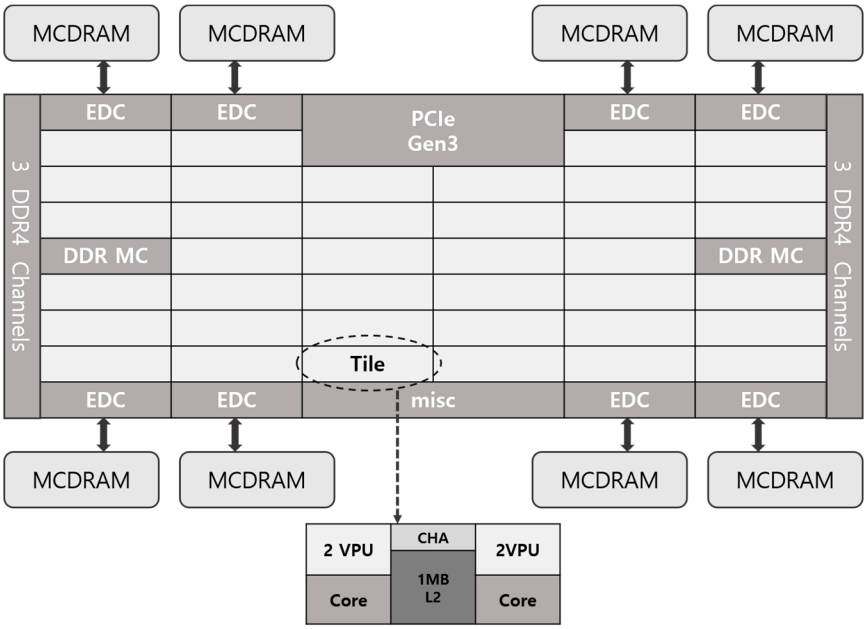


Figure 2. Diagram of Knights Landing (KNL) process architecture. Redrawn from [36], Sandia National Lab. (SNL-NM), 2017.

KNL has three main Cluster modes. Cluster mode defines various methods in which memory requests generated by the core are executed by the memory controller. To maintain cache coherency, KNL has a Distributed Tag Directory (DTD) [37]. The DTD consists of a set of Tag Directories (TDs) for

each tile that identifies the state and location of the chips in the cache line. For every memory address, the hardware can identify the TD of that address as a hash function. There are three Cluster modes in KNL; All-to-All, Quadrant, and Sub-NUMA Clustering-4 (SNC-4) modes. In All-to-All mode, memory addresses are evenly distributed across all TDs of a chip, and this may result in a long waiting time due to cache hits and cache misses. Quadrant mode divides the tile into four parts called quadrants. The chip is divided into quadrants but mapped to a single Non-Uniform Memory Access (NUMA) node. Memory addresses served by a memory controller in a quadrant are guaranteed to be mapped only to TDs contained in that quadrant. SNC-4 mode divides the chip into quadrants, and each quadrant is mapped to a different NUMA node. In this mode, NUMA-aware software can pin software threads to the same quadrant that contains the TD and accesses NUMA-local memory. SNC-4 mode has a memory controller and TD in the same area and has a shorter path than Quadrant mode. Because Memory controller are physically close to each other, this mode has the lowest latency provided that communication stays within a NUMA domain. It is recommended that users set the Cluster mode according to the characteristics of the application. An appropriate mode should be set for the lowest latency with the cache and the largest communication bandwidth.

KNL has two types of memory, MCDRAM and Double Data Rate 4 (DDR4) [38]. Multi-Channel Dynamic Random-Access Memory (MCDRAM) is an On-package High-Bandwidth Memory. MCDRAM has about five times the bandwidth of reading and writing than DDR4. MCDRAM has three Memory modes: Cache, Flat, and Hybrid. In Cache mode, the entire MCDRAM is used as a cache. In Flat mode, both MCDRAM and DDR4 memory are mapped to the same system address space. Hybrid mode is divided so that a part of MCDRAM is used as a cache, and the rest is used like Flat mode.

4. Python-Based MPI Collective Communication Library

This section describes how we build the Python-based MPI collective communication library considering features of many-core architecture and MCDRAM. We used the Memkind library and the Cython Wrapping method to use high-performance memory MCDRAM in Python environment. In addition, we discuss how to bind Docker Container to core and memory when performing experiments in Docker environment.

4.1. Memkind Library for Using MCDRAM

In order to use MCDRAM, a high-performance memory in the Intel Xeon Phi Knights Landing processor, we use the Memkind library, which is a user extensible heap manager built on top of jemalloc [12]. The Memkind library can control memory characteristics and partition the heap between memory types. Memkind is implemented in user space. It allows the user to manage new memory, such as MCDRAM, without modifying the application software or operating system.

The Memkind library can be generalized to any NUMA architecture, but in the case of Knights Landing processors, it is mainly used to manually allocate to MCDRAM using special assignments for C/C++. Figure 3 presents how Memkind library allocates memory through two different interfaces, hbwmalloc and memkind. Both interfaces use the same backend. The memkind interface is used as a general interface, and the hbwmalloc interface is used for high bandwidth memory like MCDRAM. The memkind interface uses the memkind.h file, and the hbwmalloc interface uses the hbwmalloc.h file. In this paper, the hbwmalloc interface is used as an experiment using MCDRAM.

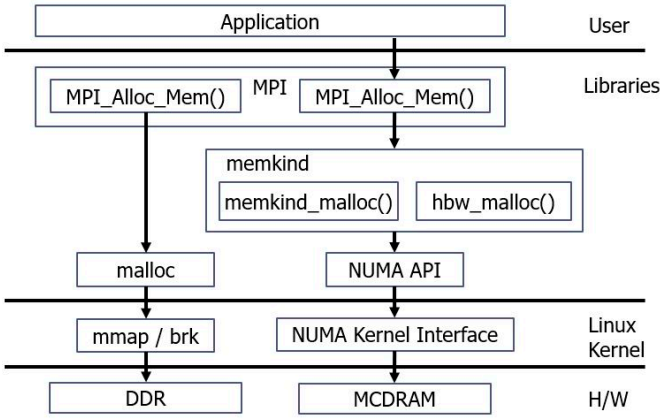


Figure 3. Architecture difference between general memory allocation and Multi-Channel Dynamic Random Access Memory (MCDRAM) memory allocation using Memkind.

In general, when dynamically allocating memory in C and C++ languages, it is allocated using functions, such as malloc, realloc, and calloc. In addition, the free function is used to deallocate dynamic memory. For dynamic memory allocation using the hbwmalloc interface of the Memkind library, memory can be allocated in MCDRAM using the hbw_malloc, hbw_realloc, and hbw_calloc functions. Using the hbw_free function can deallocate allocated memory. To implement code in C and C++ languages using the hbwmalloc interface in a Linux environment, we use gcc with -lmemkind option when compiling the codes.

4.2. Wrapping Memkind

This subsection describes how we use the KNL processor's MCDRAM in a Python environment. Numpy [39–41] and Memkind libraries are used to perform Python MPI communication using MCDRAM. Numpy is a Python library for mathematical and scientific operations that makes it easy to process matrices or large multidimensional arrays. Wrapping is implemented in Python using Cython [13,14] to use Memkind library in Python environments. Figure 4 shows the Wrapping method with Cython is written in the following four steps.

- Step 1. Implement C code that allocates memory using Memkind.
- Step 2. The Cython file has .pyx extension. Wrap the C code implemented in Step 1 using Cython code.
- Step 3. The Cython code must be compiled differently from Python. Implement a setup file to perform compilation.
- Step 4. Import the C module defined in Cython.pyx and setup.py into the Python MPI code.

In the first step, hbw_malloc.c implements the code that allocates memory using the hbw_malloc function from the Memkind library. In the second step, Cython.pyx is defined to wrap the array allocated memory with the hbw_malloc function in the first step into an array of Numpy. To receive C code from a Python object, Cython.pyx needs to get a pointer to the PyObject structure. This step defines the arrays used to exchange data. We use PyArrayObject when defining an array. PyArrayObject is a C API provided by Numpy. Numpy provides a C API to access array objects for system extensions and use in other routines. PyArrayObject contains all the information necessary for a C structured array. We use the PyArray_SimpleNewFromData function on PyArrayObject and create a Python array wrapper for C [42]. In the third step, the Python code setup.py is implemented using the distutils package provided by Cython to compile the Cython.pyx file. The distutils package supports building and installing additional modules in Python. To compile the Cython.pyx file, setup.py contains all compile-related information. The setup.py uses Cython compiler to compile Cython.pyx file into C extension file. After that, the C extension file is converted to the .so or .pyd extension file by the C

compiler, and .so file is used as a Python module. Finally, the fourth step is to implement the Python MPI code to exchange the actual data. Implementing the MPI code defines the Numpy used for data communication. In defining Numpy, the hbw_malloc module defined in Cython.pyx is used for the array data part of Numpy. MPI communication is performed with Python wrapped with Memkind and Numpy. Therefore, through this process, data exchange is performed using MCDRAM.

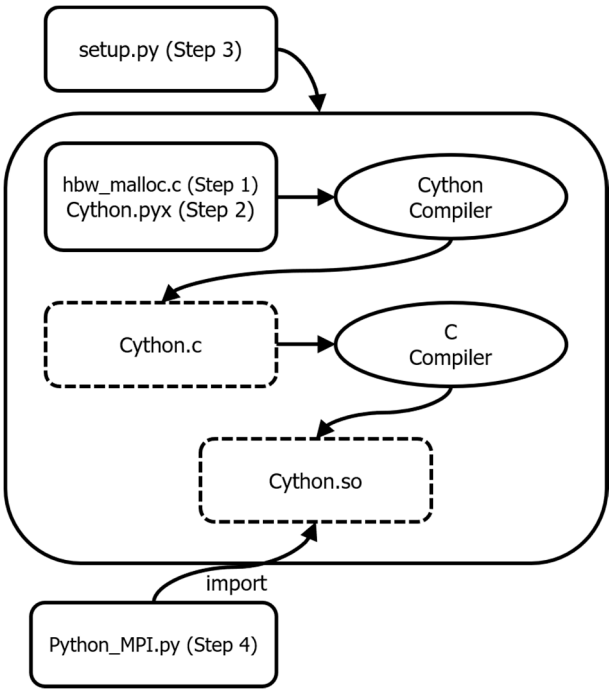


Figure 4. Memkind Cython Wrapping implementation architecture.

4.3. Binding Memory and Core to Docker

This subsection describes how we conduct experiments in a Docker environment considering a distributed environment. Docker is an open source that automates the task of placing Linux applications into software called Containers. A Container is a standard software unit that packages code and all dependencies. This allows applications to run quickly and reliably from one computing environment to another.

In the experiment of this study, to analyze the optimal MPI communication performance in KNL, we experiment according to Memory mode and Cluster mode. In the experiment, Memory mode uses Flat mode and Cache mode, and Cluster mode uses Quadrant mode and SNC-4 mode. In Quadrant mode the chip is divided into four virtual quadrants but is exposed to the operating system as a single NUMA node [33]. In Quadrant mode, unlike SNC-4 mode, the chip is not divided into multiple NUMA nodes. In Flat mode, MCDRAM is used as an operating system's managed addressable memory. Therefore, MCDRAM is exposed as a separate NUMA node. As shown in Figure 5, DDR4 and Core are NUMA node 0, and MCDRAM is NUMA node 1. Therefore, when Flat mode is selected in Quadrant mode, there are two NUMA domains. Thus, in Quadrant Mode, there is no need to allocate NUMA Nodes to two Docker Containers or four Docker Containers.

On the other hand, in the SNC-4 mode, each quadrant of the chip is exposed as a separate NUMA node to the operating system. SNC-4 mode looks like 4 sockets in succession because each quadrant appears as a separate NUMA domain to the operating system. Each node divides the core and DDR4 into quarters and appears grouped into four nodes. In addition, each node divides the MCDRAM into quarters and adds four NUMA nodes. Therefore, a total of 8 NUMA nodes are exposed to the operating system. The divided NUMA nodes have a distance between nodes. For example, in Figure 6, node 0 is closer to node 4 than other MCDRAM nodes (node5, node6, node7).

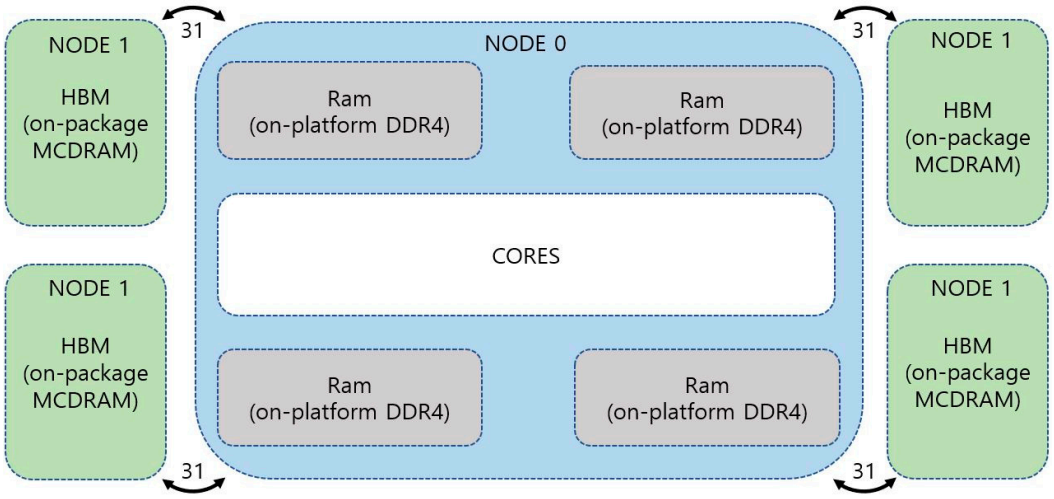


Figure 5. Non-Uniform Memory Access (NUMA) configuration in Quadrant mode with High-Bandwidth Memory (HBM) in Flat mode.

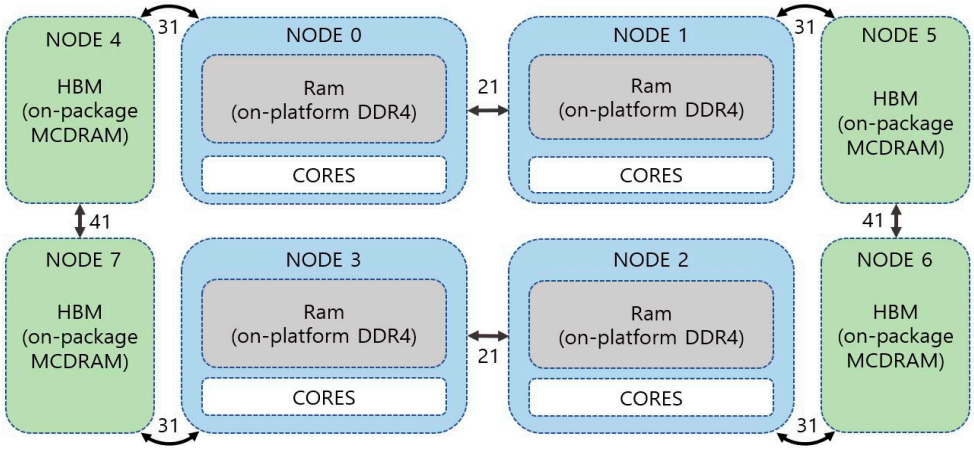


Figure 6. NUMA configuration in Sub-NUMA Clustering-4 (SNC-4) mode with HBM in Flat mode.

In order to obtain optimized performance in SNC-4 mode, code modification or NUMA environment control is required. In SNC-4 mode, when the path between the MCDRAM node and DDR4 + Core node is shortly allocated, the waiting time is reduced, and the performance is improved. Therefore, we assign a Container to two adjacent nodes like the architecture in Figure 7. The first Container binds to node 0 (core, DDR4) and node 4 (MCDRAM). The second Container binds node 1 and node 5, the third Container binds node 2 and node 6, and the fourth Container binds node 3 and node 7. When creating a Docker Container, we declare the privileged option so that the Containers can access the Host's device. We set up a port to communicate between each Container. For the Host environment, we use the same binding policy of core and memory in SNC-4 mode.

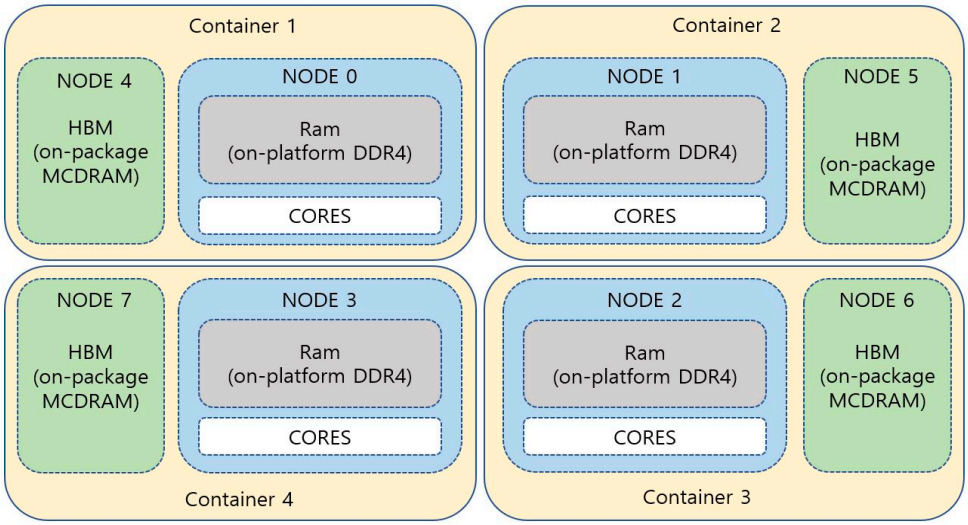


Figure 7. Four Docker Containers Architecture bound to NUMA node in SNC-4 mode.

5. Performance Evaluation

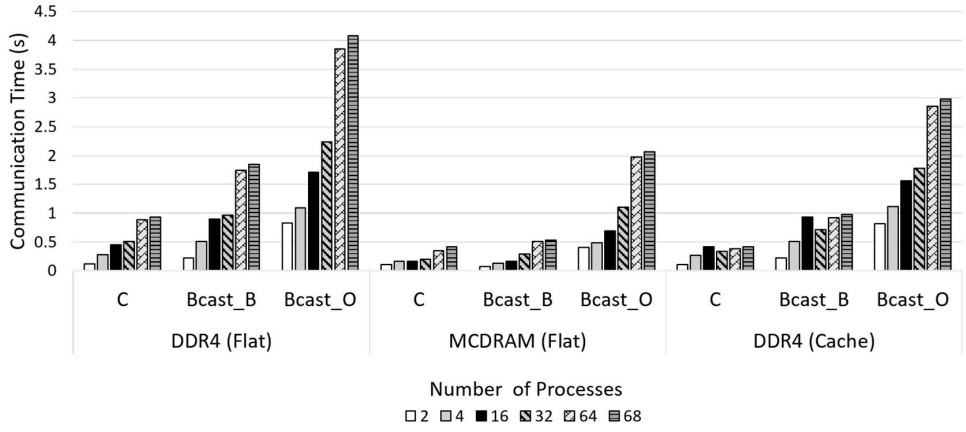
In this section, we compare and analyze MPI communication performance in high-performance memory and Python environment through experiments. The server used in the experiment consists of a KNL processor with 68 cores, 16 GB of MCDRAM, and 96 GB of DDR4. For software (SW) environment, we used Linux (Kernel Version 5.2.8) and Intel MPI (Version 2017 Update 3). Due to the GIL feature of Python, experiments were conducted by exchanging data using MPI, which is a multi-processing method instead of multi-threading. We tested collective communication of Broadcast, Gather, and AllReduce methods, which are widely used as MPI communication methods in parallel and distributed deep learning. Python's MPI communication has two methods: Object and Buffer communication [43]. Object communication provides Pickle-based communication to conveniently communicate objects in Python. Object communication uses all lowercase methods, like bcast(), gather(), and allreduce(). Buffer communication is an efficient communication method for array data communication, such as NumPy. Buffer communication uses uppercase, like Bcast(), Gather(), and AllReduce(). The Pickle base of object communication converts object data into byte streams and byte streams back into object data [44]. On the other hand, buffer communication allows direct access to object data using the buffer interface [45]. The objects supported by the buffer interface are byte and array objects. In this study, buffer communication can be used because the Numpy array is used in the experiment. Object communication is simple to use but has an overhead compared to buffer communication, which does not require a conversion process.

In Sections 5.1–5.3, to evaluate the proposed scheme, the Cluster mode of the KNL processor was tested in Quadrant mode. Memory mode was set in Flat mode and Cache mode for comparison. Different combinations of DDR4 and MCDRAM were compared by changing the number of processes in C and Python environments. In addition, to see the Python communication performance characteristic, we used the Buffer and Object communication methods. In order to consider the distributed environment, we conducted an experiment on distributed communication in Docker's virtual environment. In Section 5.4, we compare Quadrant mode and SNC-4 mode among Cluster modes of KNL processor.

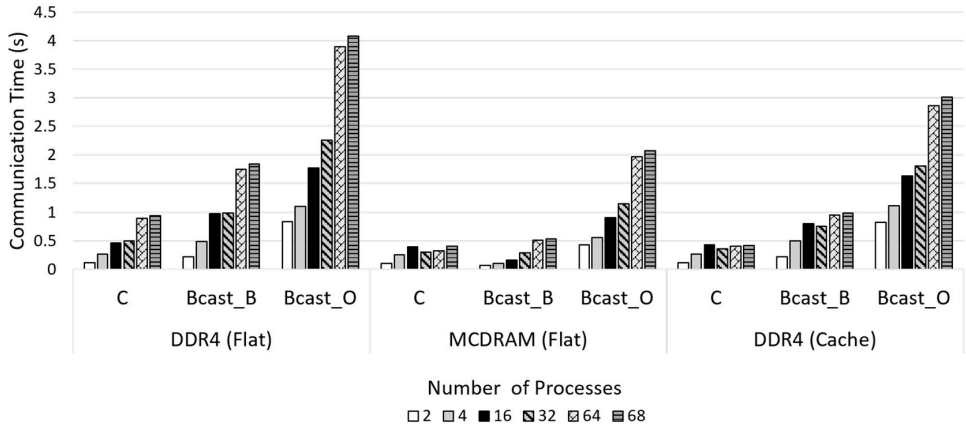
5.1. Broadcast

In the Broadcast experiment, the data type was Integer and the size of the transferred data was 256 MB. Figure 8 graph shows the result of Broadcast communication time. In the experiment graph of Figure 8, C represents MPI communication in C environment, Bcast_B represents MPI

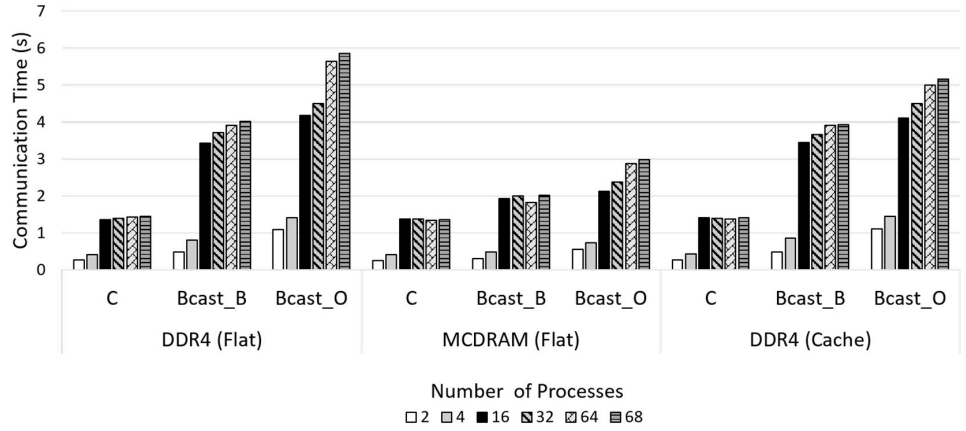
buffer communication in Python environment, and Bcast_O represents MPI object communication in Python environment.



(**a**)

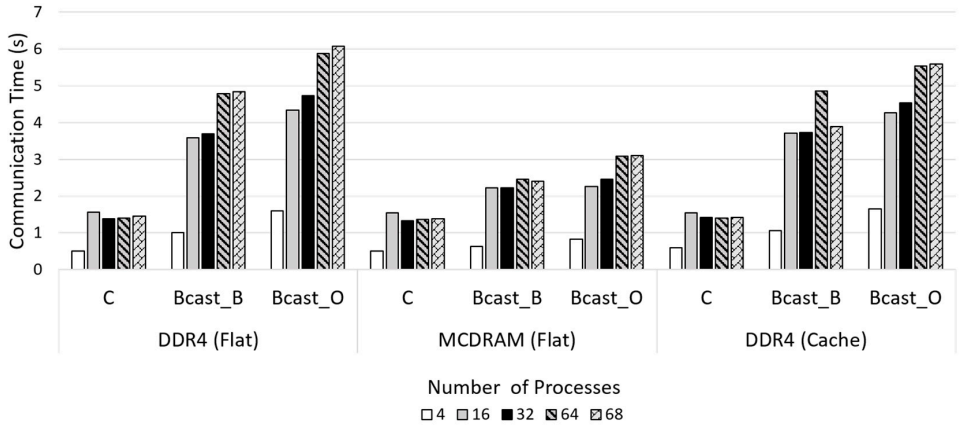


(**b**)



(**c**)

Figure 8. *Cont.*



(**d**)

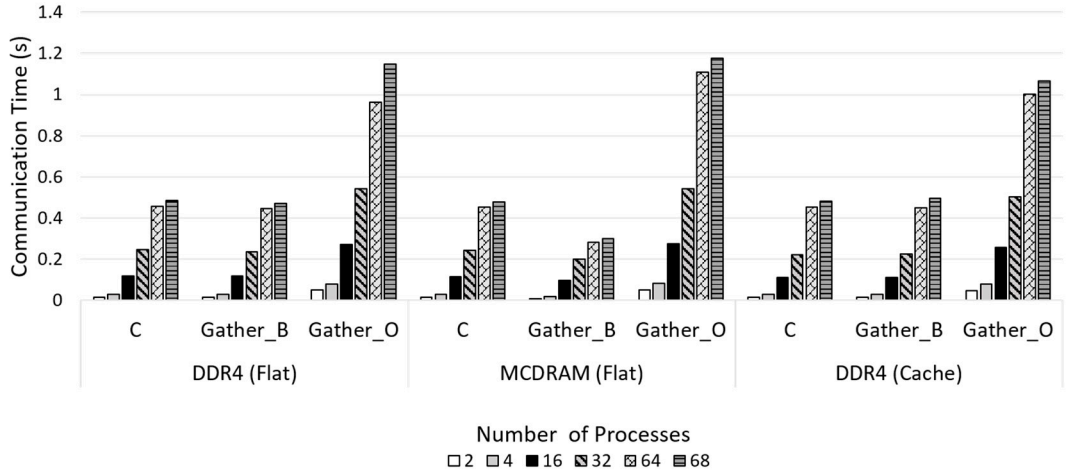
Figure 8. Broadcast communication time results according to the experiment environment: (**a**) Host; (**b**) one Docker Container; (**c**) two Docker Containers; (**d**) four Docker Containers.

Figure 8 shows the result of executing Broadcast. Figure 8a is the Host environment, Figure 8b is the one Docker Container environment, Figure 8c is the two Docker Containers environment, and Figure 8d is the four Docker Containers environment. In all environments, communication using MCDRAM in Flat mode showed the fastest communication time. The case of using DDR4 in Cache mode was second fastest. Finally, the slowest communication time was shown when DDR4 was used in Flat mode. The method proposed in this paper is Bcast_B, which performs Python buffer communication when using MCDRAM in Flat mode. When using only DDR4 in Flat mode, Bcast_B showed slower time than C. However, in the experiment environment of Host and one Container, the communication time of Bcast_B and C was similar in MCDRAM of Flat mode. Bcast_B communication in MCDRAM of Flat mode was confirmed to improve performance of up to 487% compared to Bcast_B communication in DDR4 of Flat mode. This best case was shown when 16 processes were used in one Docker Container. In addition, the performances of the Host (Figure 8a) and one Container (Figure 8b) environments were similar. This result confirmed Docker's claim that the performance difference between the main Host and Docker's Container is not significant because Docker directly uses the Host's resources. However, as the number of Containers increased in MCDRAM of Flat mode, Bcast_B showed slower communication performance compared to C. Through this result, we confirmed that multiple Docker Containers are relatively less efficient than communication in the Host environment.

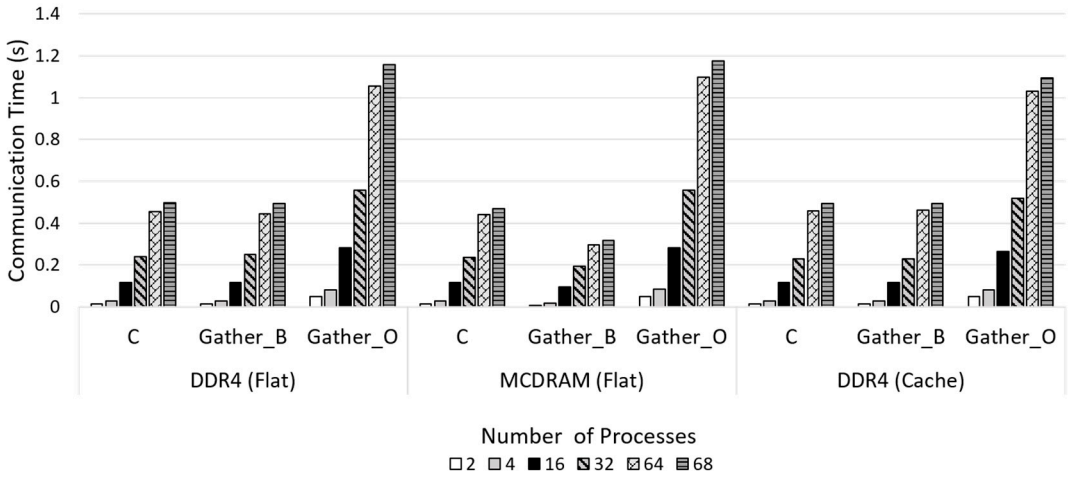
As mentioned in the background of the previous section, in Cache mode, MCDRAM is used as a cache. MCDRAM cache is treated as a Last Level Cache (LLC), which is located between L2 cache and addressable memory in the memory hierarchy of Knights Landing processors. Therefore, it can be seen that the Bcast_B communication in Cache mode is up to 88% faster than the Bcast_B communication in DDR4 of Flat mode.

5.2. Gather

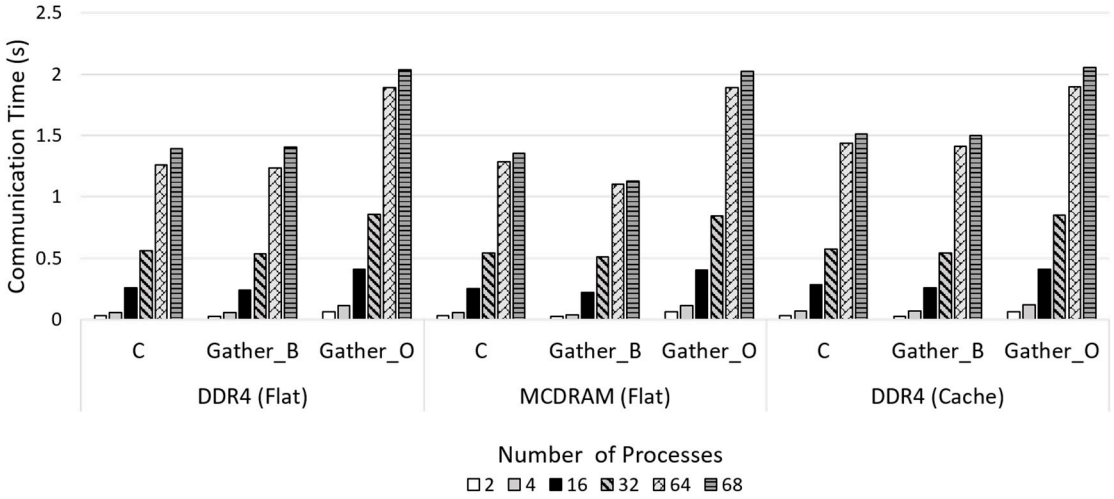
In the Gather experiment, the data type was Integer and the size of transferred data was 16 MB. Figure 9 shows the results of Gather communication times. Like the Broadcast experiment, in the Figure 9 graphs, C represents MPI communication in C environment, Gather_B represents MPI buffer communication in Python environment, and Gather_O represents MPI object communication in Python environment.



(**a**)

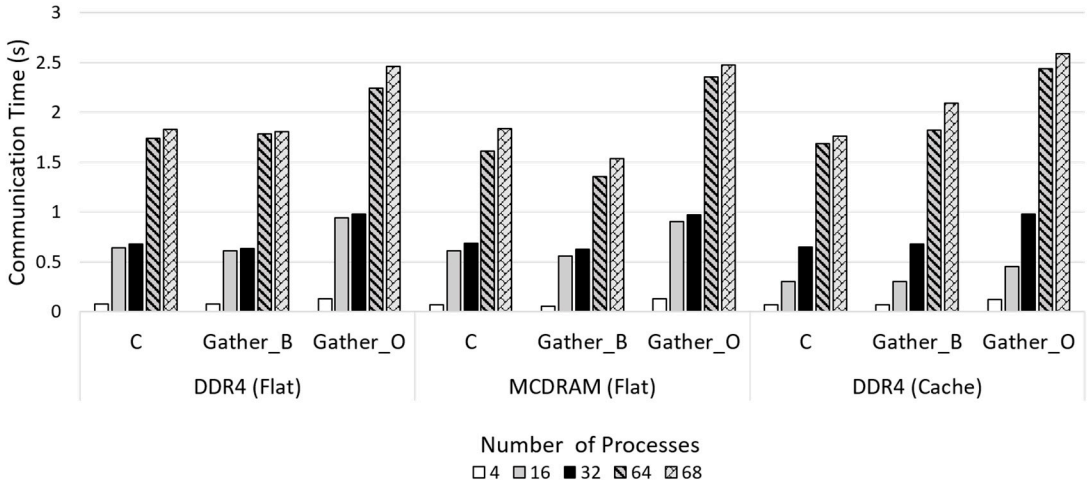


(**b**)



(**c**)

Figure 9. *Cont.*



(**d**)

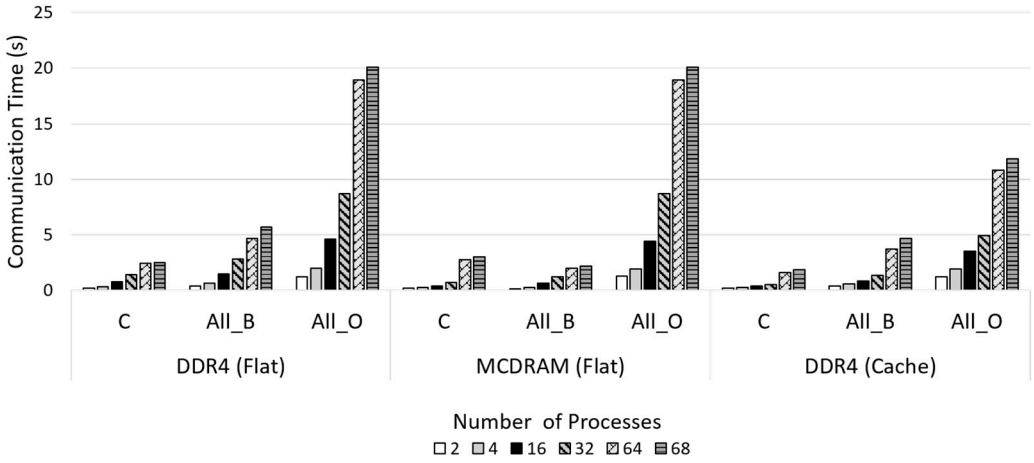
Figure 9. Gather communication time results according to the experiment environment: (**a**) Host; (**b**) one Docker Container; (**c**) two Docker Containers; (**d**) four Docker Containers.

Figure 9 is the result of performing Gather communication. Figure 9a is the Host environment, Figure 9b is the one Docker Container environment, Figure 9c is the two Docker Containers environment, and Figure 9d is the four Docker Containers environment. Similar to the Broadcast experiment in Figure 8, communication in MCDRAM of Flat mode was the fastest and DDR4 of Flat mode was the slowest. In the Gather experiment, the data size is smaller than that of Broadcast and AllReduce. Gather communication collects data from many processes in a single process. This feature of Gather communication generates Out Of Memory when the data size is large. The reason that the performance comparison between experiments seems small is that the data size is small. The feature of this experiment is that the case of Gather_B in MCDRAM of Flat mode shows faster communication performance than C.

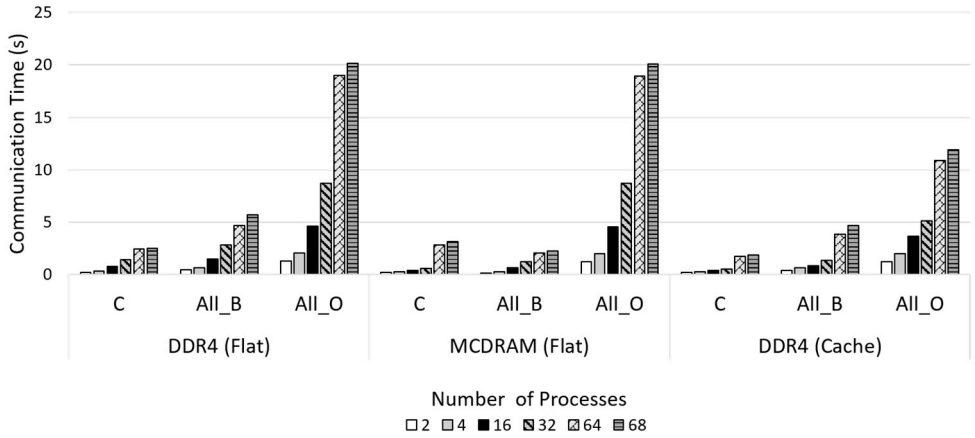
It was confirmed that the Gather_B communication in MCDRAM of Flat mode has a performance improvement of up to 65% compared to the C communication in MCDRAM of Flat mode. In addition, the Gather_B communication in MCDRAM of Flat mode was able to confirm up to 74% performance improvement over the Gather_B communication in DDR4 of Flat mode. It can be seen that Gather_B communication in Cache mode is up to 13% faster than Gather_B communication in DDR4 of Flat mode. Like Broadcast, the Host environment (Figure 9a) and the one Container environment (Figure 9b) showed similar performance. In addition, poor communication performance was confirmed in the increase of the number of Containers.

5.3. AllReduce

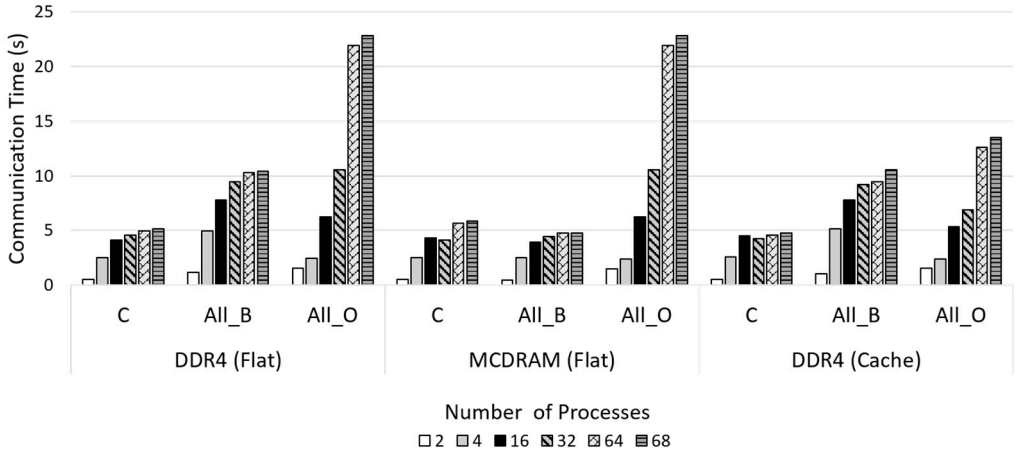
In the AllReduce experiment, the data type was Integer and the size of transferred data was 256MB. Figure 10 graph shows the result of AllReduce communication time. Like the previous experiments, in the graphs of Figure 10, C indicates MPI communication in C environment, All_B indicates MPI buffer communication in Python environment, and All_O indicates MPI object communication in Python environment.



(**a**)

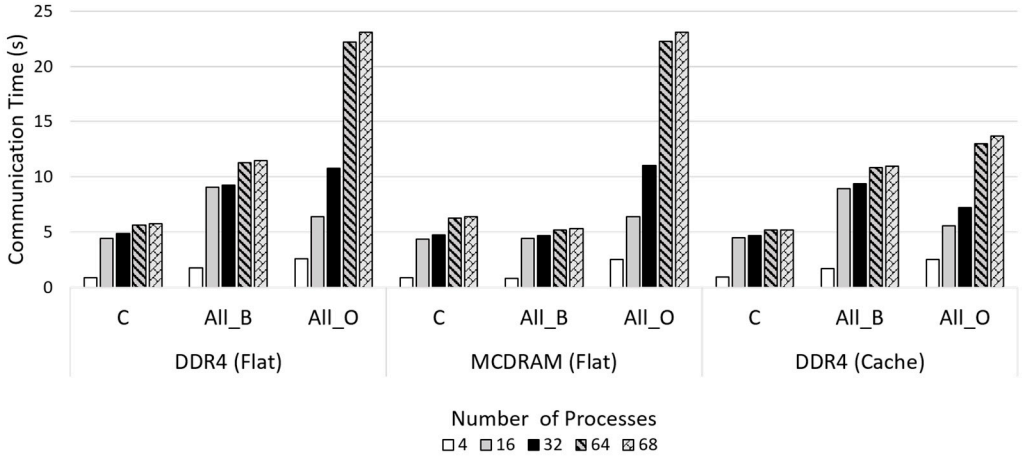


(**b**)



(**c**)

Figure 10. *Cont.*



(**d**)

Figure 10. AllReduce communication time results according to the experiment environment: (**a**) Host; (**b**) one Docker Container; (**c**) two Docker Containers; (**d**) four Docker Containers.

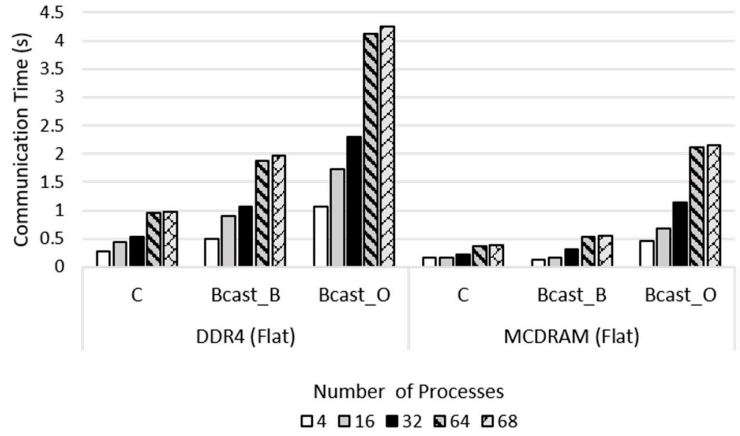
Figure 10 is the result of executing AllReduce. Figure 10a is the Host environment, Figure 10b is the one Docker Container environment, Figure 10c is the two Docker Containers environment, and Figure 10d is the four Docker Containers environment. In the experiment, the communication result between the experimental environments is the same as the previous experiments. With DDR4 in Flat mode, All_B showed slower performance than C. However, in MCDRAM of Flat mode, the communication performance of All_B and C was similar. A special feature of the AllReduce experiment is seen in Python's object communication. Most of the previous communication experiments showed the fastest performance in MCDRAM of Flat mode. In AllReduce, object communication All_O in Cache mode showed faster performance than object communication All_O in MCDRAM of Flat mode. This result seems to be due to the feature of AllReduce, which combines values from all processes and distributes the results to all processes. Unlike the previous experiments, AllReduce has a data operation task. It seems that the utilization of cache has increased for computation. Therefore, it was confirmed that object communication All_O in Cache mode improved the performance by up to 76% compared to object communication All_O in MCDRAM of Flat mode.

The All_B communication in MCDRAM of Flat mode showed up to 162% performance improvement over All_B communication in DDR4 of Flat mode. In Cache mode, All_B communication is up to 110% faster than DDR4 communication in Flat mode. The Host environment (Figure 10a) and one Container environment (Figure 10b) showed almost similar performance. Like the previous experiments, when the number of Containers increased, the poor communication performance was confirmed.

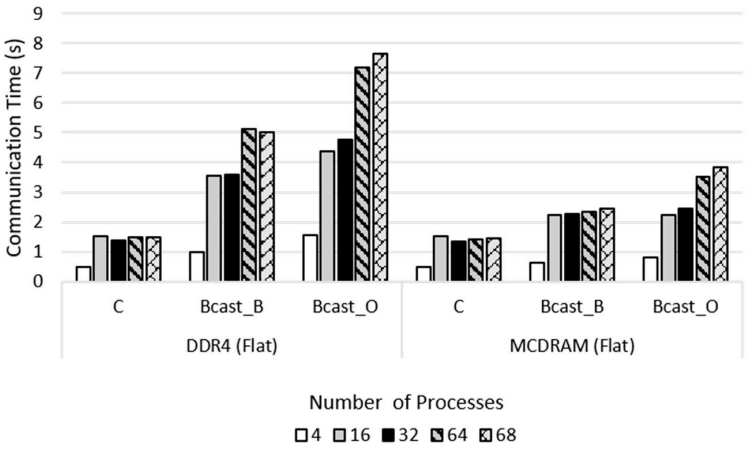
5.4. Cluster Mode

In this subsection, we compare and analyze the performance according to the Cluster mode. The previous experiments were performed in the environment where Cluster mode is Quadrant mode. In this subsection, experiments are performed in SNC-4 mode. Then, we compared and analyzed the performance differences according to Cluster mode. The data type in the experiment was Integer, and the data size of Broadcast and AllReduce was 256 MB, but the data size of Gather was 16 MB. In the experiment graph of Figure 11, C represents MPI communication in C environment; Bcast_B, Gather_B, and All_B represent MPI buffer communication in Python environment; and Bcast_O, Gather_O, and All_O represent MPI object communication in Python environment. The comparison method is the same as in the previous sections; however, the Memory mode experimented with is DDR4 in Flat

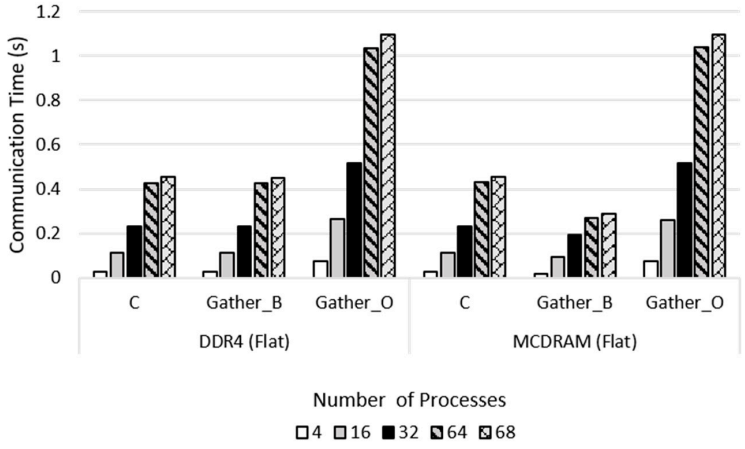
mode and MCDRAM in Flat mode. This section differs from previous ones in that we compare just two environment types, Host and four Docker Containers.



(**a**)

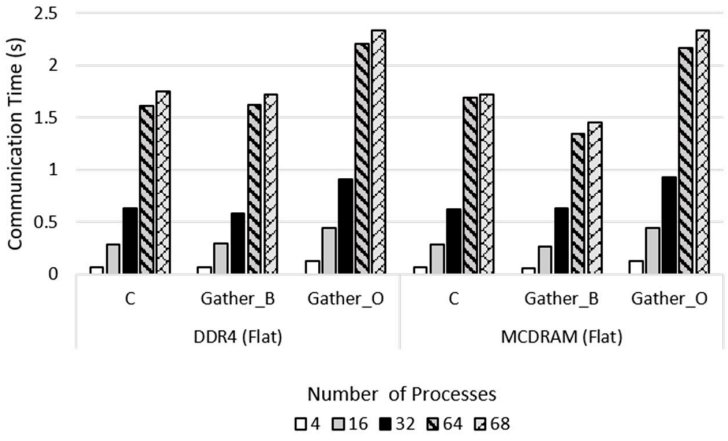


(**b**)

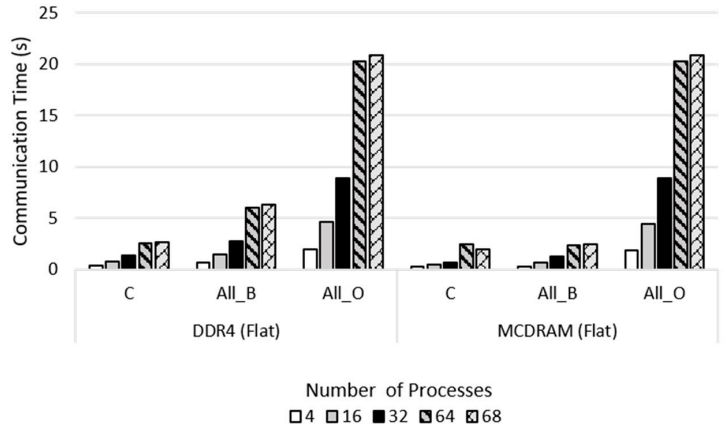


(**c**)

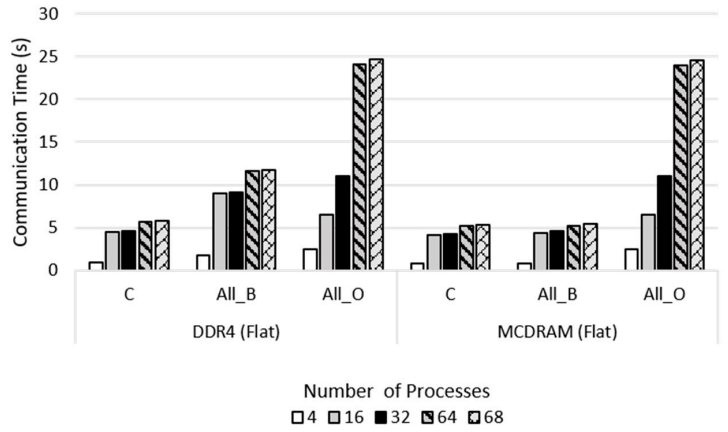
Figure 11. *Cont.*



(**d**)



(**e**)



(**f**)

Figure 11. BroadCast, Gather, and AllReduce communication time result in SNC-4 Cluster mode environment: (**a**) Broadcast Host; (**b**) Broadcast four Docker Containers; (**c**) Gather Host; (**d**) Gather four Docker Containers; (**e**) AllReduce Host; (**f**) AllReduce four Docker Containers.

Figure 11 is the experiment result in SNC-4 Cluster mode environment. Figure 11a is Broadcast in Host environment, Figure 11b is Broadcast in four Docker Containers environment, Figure 11c is Gather in Host environment, Figure 11d is Gather in four Docker Containers environment, Figure 11e

is Host environment In AllReduce, and Figure 11f is AllReduce in four Docker Containers environment. As explained in the background of the previous section, SNC-4 mode divides the chip into quadrants, and each quadrant is mapped to a NUMA node. In this mode, SNC-4 mode has a memory controller and TD in the same area and has a shorter path than Quadrant mode. For this reason, it was expected that SNC-4 mode might perform better than Quadrant mode. However, the experimental results showed little difference in performance between SNC-4 mode and Quadrant mode. Due to the characteristics of the algorithms of Broadcast, Gather, and AllReduce performed in this paper, it seems that the characteristics of SNC-4 Cluster mode cannot be utilized. In SNC-4 mode, MCDRAM NUMA node and DDR4+Core node are physically close. SNC-4 mode shows the lowest latency when communication is maintained within the NUMA domain. But, if in SNC-4 mode cache traffic crosses NUMA boundaries, this path is longer than in the Quadrant mode. In this experiment, MPI collective communication creates a graph topology with a distributed method, and each node defines a neighbor. Therefore, due to this MPI communication characteristic, it seems difficult to utilize the advantages of SNC-4 mode when the process to be communicated is not in the same domain. According to our experiment, MPI communication in Quadrant mode is effective, which can maintain transparency without code modification.

6. Summary of Experimental Results

In this section, the results of the experiments performed in this paper are summarized. Tables 1–4 compare the performance improvement rate of Python communication and C communication in the Host and four Docker Containers environments. Table 1 shows the ratio of DDR4 and MCDRAM communication times when performing Python buffer communication in Flat mode. Table 2 is a table comparing the performance of C communication in Flat mode. Tables 3 and 4 are tables comparing the experimental results in the four Docker Containers environment. Table 5 compares the communication time according to the Host environment and the increase of Docker Containers. Tables 6 and 7 compare the communication performance according to the Cluster mode. All experiments in the previous section increase the number of processes. However, in this summary, the communication time is compared in the number of processes with the best performance improvement.

Table 1. Comparison of Python buffer communication time between Double Data Rate 4 (DDR4) and MCDRAM in Host environment (Flat mode).

	DDR4/MCDRAM (%)	Number of Processes
Broadcast	545	16
Gather	159	64
AllReduce	262	68

Table 2. Comparison of C communication time between DDR4 and MCDRAM in Host environment (Flat mode).

	DDR4/MCDRAM (%)	Number of Processes
Broadcast	271	16
Gather	103	16
AllReduce	201	32

Table 3. Comparison of Python buffer communication time between DDR4 and MCDRAM in four Docker Containers environment (Flat mode).

	DDR4/MCDRAM (%)	Number of Processes
Broadcast	202	68
Gather	131	64
AllReduce	217	64

Table 4. Comparison of C communication time between DDR4 and MCDRAM in four Docker Containers environment (Flat mode).

	DDR4/MCDRAM (%)	Number of Processes
Broadcast	105	68
Gather	108	64
AllReduce	102	32

Table 5. Communication time according to the increase in the number of Docker Containers.

	Broadcast (s)	Gather (s)	AllReduce (s)
Host	0.285	0.194	1.261
One Container	0.288	0.199	1.271
Two Containers	1.991	0.508	4.447
Four Containers	2.231	0.628	4.677

Table 6. Communication time between Quadrant mode and SNC-4 mode in Host environment.

	Quadrant (s)	SNC-4 (s)
Broadcast	0.525	0.553
Gather	0.301	0.297
AllReduce	2.168	2.421

Table 7. Communication time of Quadrant mode and SNC-4 mode in four Docker Containers environment.

	Quadrant (s)	SNC-4 (s)
Broadcast	2.398	2.456
Gather	1.538	1.531
AllReduce	5.342	5.361

In Table 1, when Broadcast performed MPI collective communication with 16 processes, it was confirmed that MCDRAM performance improved 445% over DDR4. In addition, Gather improved the performance by 59% in 64 processes, and AllReduce confirmed performance improvement by 162% in 68 processes. As can be seen in Table 2, Broadcast showed 171% performance improvement in 16 processes, Gather showed 3% performance improvement in 16 processes, and, finally, AllReduce showed 101% performance improvement in 32 processes. From Tables 1 and 2, it can be confirmed that the rate of performance improvement in the same Host experiment environment is greater in Python buffer communication than in C communication. Similarly, in Tables 3 and 4, the Python buffer communication in the four Containers environment showed a higher performance improvement rate than with C communication.

Table 5 shows the communication time results of Python buffer communication in 32 processes and MCDRAM in Flat mode. In Table 5, the Host environment and one Container show almost the same communication time. In addition, when the number of Containers increases, it can be seen that all collective communication time increases. In Table 5, it was confirmed that there was little difference in performance between Host and one Docker, as claimed by Docker. However, multiple Containers show relatively less efficient communication than communication in the Host environment.

The experimental results in Tables 6 and 7 are the communication time results of the Python buffer communication in 68 processes and MCDRAM in Flat mode. In Tables 6 and 7, Quadrant mode and SNC-4 mode are similar, or SNC-4 mode shows slow communication time. This experimental result shows that the advantages of SNC-4 mode cannot be utilized. Due to the MPI communication feature, it seems that the advantages of SNC-4 mode cannot be utilized when the communication process is not in the same domain. As a result, in MPI communication, the Quadrant mode that can maintain transparency may be more efficient than the SNC-4 mode.

Our performance evaluation can be summarized by the following three points:

- The combination of MCDRAM usage and Python buffer communication has greatly improved the performance of Python's MPI collective communication. This result shows that MCDRAM was efficiently used in the Python environment by wrapping Memkind library. And it was confirmed that Python's buffer communication is more efficient than Python's object communication. In addition, in Memory mode of Knights Landing Process, the use of MCDRAM in Flat mode showed better performance than Cache mode using MCDRAM as a cache (Tables 1–4).
- C communication is still fast in a distributed Docker environment. All experiments showed similar communication performance on the Host and one Docker Container environments. In this result, we confirmed Docker's claim that the performance difference between the main Host and Docker's Container is not significant because Docker's Container directly uses the Host's resources. However, multiple Containers show relatively less efficient communication than communication in the Host environment (Table 5).
- The Quadrant mode showed more efficient experimental results than the SNC-4 mode. SNC-4 mode has a low latency when communication is maintained within the NUMA node. Therefore, it is difficult to efficiently utilize the SNC-4 mode when its process of communication is not in the same domain. Due to the characteristic of MPI communication, the advantages of SNC-4 mode cannot be utilized. In Cluster mode, Quadrant mode, which ensures transparency, is an efficient method for MPI communication (Tables 6 and 7).

7. Conclusions

In this paper, we proposed a method of efficiently exchanging data when performing distributed deep learning in Python and many-core CPU environments, and we compared and analyzed the proposed method through extensive experiments in various environments. For performance evaluation, we compared performance according to KNL's Memory mode and Cluster mode and also compared performance according to the Python MPI communication method. In addition, to consider a distributed environment, we experimented in a Docker environment.

The combination of MCDRAM usage and Python buffer communication has greatly improved the performance of Python's MPI collective communication. C communication is still fast in the distributed Docker environment, but similar communication performance was also seen in the Host and one Docker Container environments. In addition, it was confirmed that multiple Docker Containers are relatively less efficient than communication in the Host environment. In the Cluster mode experiment, the Quadrant mode showed more efficient experimental results than the SNC-4 mode. Due to the characteristic of MPI communication, the advantages of SNC-4 mode cannot be utilized. In Cluster mode, Quadrant mode, which ensures transparency, is an efficient method for MPI communication.

In the future, we plan to apply to deep learning frameworks, such as Google's TensorFlow and Facebook's PyTorch, using our proposed MPI library, and further explore performance improvements.

Author Contributions: Conceptualization, J.W. and J.L.; methodology, J.W. and J.L.; software, J.W. and H.C.; writing—original draft preparation, J.W.; writing—review and editing, J.W., H.C. and J.L.; supervision, J.L.; project administration, J.L.; funding acquisition, J.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Next-Generation Information Computing Development Program (2015M3C4A7065646) and Basic Research Program (2020R1F1A1072696) through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT, GRRC program of Gyeong-gi province (No. GRRC-KAU-2020-B01, "Study on the Video and Space Convergence Platform for 360VR Services"), ITRC (Information Technology Research Center) support program (IITP-2020-2018-0-01423), the Korea Institute of Science and Technology Information (Grant No.K-20-L02-C08), and the National Supercomputing Center with supercomputing resources including technical support (KSC-2019-CRE-0101).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016.
2. Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. Tensorflow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; pp. 265–283.
3. Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; Lerer, A. Automatic differentiation in pytorch. In Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017.
4. Li, Y.; Park, J.; Alian, M.; Yuan, Y.; Qu, Z.; Pan, P.; Wang, R.; Schwing, A.; Esmaeilzadeh, H.; Kim, N.S. A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks. In Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2018), Fukuoka, Japan, 20–24 October 2018; pp. 175–188.
5. Oliphant, T.E. Python for scientific computing. *Comput. Sci. Eng.* **2007**, *9*, 10–20. [CrossRef]
6. Beazley, D. Understanding the python gil. In Proceedings of the 2010 PyCON Python Conference, Atlanta, GA, USA, 25–26 September 2010.
7. Global Interpreter Lock. Available online: https://wiki.python.org/moin/GlobalInterpreterLock/ (accessed on 20 May 2020).
8. MPI Forum. Available online: https://www.mpi-forum.org/ (accessed on 20 May 2020).
9. Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput.* **1996**, *22*, 789–828. [CrossRef]
10. MPI Tutorial. Available online: https://mpitutorial.com/ (accessed on 20 May 2020).
11. Sodani, A. Knights landing (knl): 2nd generation intel® xeon phi processor. In Proceedings of the 2015 IEEE Hot Chips 27 Symposium (HCS), Cupertino, CA, USA, 22–25 August 2015; pp. 1–24.
12. Memkind Library. Available online: https://github.com/memkind/memkind/ (accessed on 20 May 2020).
13. Cython. Available online: https://cython.org/ (accessed on 20 May 2020).
14. Behnel, S.; Bradshaw, R.; Citro, C.; Dalcin, L.; Seljebotn, D.S.; Smith, K. Cython: The best of both worlds. *Comput. Sci. Eng.* **2011**, *13*, 31–39. [CrossRef]
15. Docker. Available online: https://www.docker.com/ (accessed on 20 May 2020).
16. Ahn, S.; Kim, J.; Lim, E.; Kang, S. Soft memory box: A virtual shared memory framework for fast deep neural network training in distributed high performance computing. *IEEE Access* **2018**, *6*, 26493–26504. [CrossRef]
17. Peng, I.B.; Gioiosa, R.; Kestor, G.; Vetter, J.S.; Cicotti, P.; Laure, E.; Markidis, S. Characterizing the performance benefit of hybrid memory system for HPC applications. *Parallel Comput.* **2018**, *76*, 57–69. [CrossRef]
18. Cho, J.Y.; Jin, H.W.; Nam, D. Exploring the Performance Impact of Emerging Many-Core Architectures on MPI Communication. *J. Comput. Sci. Eng.* **2018**, *12*, 170–179. [CrossRef]
19. Li, Z.; Kihl, M.; Lu, Q.; Andersson, J.A. Performance overhead comparison between hypervisor and container based virtualization. In Proceedings of the 31st IEEE International Conference on Advanced Information Networking and Applications (AINA 2017), Taipei, Taiwan, 27–29 March 2017; pp. 955–962.
20. Zhang, J.; Lu, X.; Panda, D.K. Is singularity-based container technology ready for running MPI applications on HPC clouds? In Proceedings of the 10th International Conference on Utility and Cloud Computing (UCC 2017), Austin, TX, USA, 5–8 December 2017; pp. 151–160.
21. Zhang, J.; Lu, X.; Panda, D.K. High performance MPI library for container-based HPC cloud on InfiniBand clusters. In Proceedings of the 45th International Conference on Parallel Processing (ICPP 2016), Philadelphia, PA, USA, 16–19 August 2016; pp. 268–277.
22. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; pp. 248–255.
23. Dean, J.; Corrado, G.; Monga, R.; Chen, K.; Devin, M.; Mao, M.; Ranzato, M.; Senior, A.; Tucker, P.; Yang, K.; et al. Large scale distributed deep networks. In Proceedings of the 26th Conference on Neural Information Processing Systems (NIPS), Lake Tahoe, NV, USA, 3–8 December 2012; pp. 1223–1231.

24. Huang, Y.; Cheng, Y.; Bapna, A.; Firat, O.; Chen, D.; Chen, M.; Lee, H.; Ngiam, J.; Le, Q.V.; Wu, Y. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In Proceedings of the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, BC, Canada, 8–14 December 2019; pp. 103–112.
25. Kim, Y.; Lee, J.; Kim, J.S.; Jei, H.; Roh, H. Efficient multi-GPU memory management for deep learning acceleration. In Proceedings of the 2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W), Trento, Italy, 3–7 September 2018; pp. 37–43.
26. Kim, Y.; Lee, J.; Kim, J.S.; Jei, H.; Roh, H. Comprehensive techniques of multi-GPU memory optimization for deep learning acceleration. *Clust. Comput.* **2019**, *23*, 2193–2204. [CrossRef]
27. Kim, Y.; Choi, H.; Lee, J.; Kim, J.S.; Jei, H.; Roh, H. Efficient Large-Scale Deep Learning Framework for Heterogeneous Multi-GPU Cluster. In Proceedings of the 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W), Umeå, Sweden, 16–20 June 2019; pp. 176–181.
28. Kim, Y.; Choi, H.; Lee, J.; Kim, J.S.; Jei, H.; Roh, H. Towards an optimized distributed deep learning framework for a heterogeneous multi-GPU cluster. *Clust. Comput.* **2020**, *23*, 2287–2300. [CrossRef]
29. Heigold, G.; McDermott, E.; Vanhoucke, V.; Senior, A.; Bacchiani, M. Asynchronous stochastic optimization for sequence training of deep neural networks. In Proceedings of the 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Florence, Italy, 4–9 May 2014; pp. 5587–5591.
30. Sergeev, A.; Del Balso, M. Horovod: Fast and easy distributed deep learning in TensorFlow. *arXiv* **2018**, arXiv:1802.05799.
31. CPython. Available online: https://docs.python.org/3/ (accessed on 20 May 2020).
32. Zhang, C.; Yuan, X.; Srinivasan, A. Processor affinity and MPI performance on SMP-CMP clusters. In Proceedings of the 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), Atlanta, GA, USA, 19–23 April 2010; pp. 1–8.
33. Neuwirth, S.; Frey, D.; Bruening, U. Communication models for distributed intel xeon phi coprocessors. In Proceedings of the 2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS), Melbourne, Australia, 14–17 December 2015; pp. 499–506.
34. Lee, C.; Lee, J.; Koo, D.; Kim, C.; Bang, J.; Byun, E.; Eom, H. Empirical Analysis of the I/O Characteristics of a Highly Integrated Many-Core Processor. In Proceedings of the 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C), Washington, DC, USA, 17–21 August 2020; pp. 1–6.
35. Sodani, A.; Gramunt, R.; Corbal, J.; Kim, H.S.; Vinod, K.; Chinthamani, S.; Hutsell, S.; Agarwal, R.; Liu, Y.C. Knights landing: Second-generation intel xeon phi product. *IEEE Micro* **2016**, *36*, 34–46. [CrossRef]
36. Agelastos, A.M.; Rajan, M.; Wichmann, N.; Baker, R.; Domino, S.P.; Draeger, E.W.; Anderson, S.; Balma, J.; Behling, S.; Berry, M.; et al. *Performance on Trinity Phase 2 (a Cray XC40 Utilizing Intel Xeon Phi Processors) with Acceptance Applications and Benchmarks*; Sandia National Lab. (SNL-NM): Albuquerque, NM, USA, 2017.
37. Vladimirov, A.; Asai, R. *Clustering Modes in Knights Landing Processors: Developer's Guide*; Colfax International: Santa Clara, CA, USA, 2016.
38. Jeffers, J.; Reinders, J.; Sodani, A. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*; Morgan Kaufmann: Burlington, MA, USA, 2016.
39. Numpy. Available online: https://numpy.org/ (accessed on 20 May 2020).
40. Oliphant, T.E. A Guide to NumPy. Available online: https://web.mit.edu/dvp/Public/numpybook.pdf (accessed on 23 September 2020).
41. Walt, S.V.D.; Colbert, S.C.; Varoquaux, G. The NumPy array: A structure for efficient numerical computation. *Comput. Sci. Eng.* **2011**, *13*, 22–30. [CrossRef]
42. SciPy. Available online: https://scipy.org/ (accessed on 20 May 2020).
43. MPI for Python. Available online: https://mpi4py.readthedocs.io/en/stable/ (accessed on 20 May 2020).
44. Pickle-Python Object Serialization. Available online: https://docs.python.org/3/library/pickle.html/ (accessed on 20 May 2020).
45. Buffer Protocol. Available online: https://docs.python.org/3/c-api/buffer.html/ (accessed on 20 May 2020).