PAPER *Special Section on Parallel and Distributed Computing and Networking*

Skew-Aware Collective Communication for MapReduce Shuffling

Harunobu DAIKOKU[†a)], **Hideyuki KAWASHIMA**[††], ***and*** **Osamu TATEBE**[†], ***Nonmembers***

SUMMARY This paper proposes and examines the three in-memory shuffling methods designed to address problems in MapReduce shuffling caused by skewed data. Coupled Shuffle Architecture (CSA) employs a single pairwise all-to-all exchange to shuffle both blocks, units of shuffle transfer, and meta-blocks, which contain the metadata of corresponding blocks. Decoupled Shuffle Architecture (DSA) separates the shuffling of meta-blocks and blocks, and applies different all-to-all exchange algorithms to each shuffling process, attempting to mitigate the impact of stragglers in strongly skewed distributions. Decoupled Shuffle Architecture with Skew-Aware Meta-Shuffle (DSA w/ SMS) autonomously determines the proper placement of blocks based on the memory consumption of each worker process. This approach targets extremely skewed situations where some worker processes could exceed their node memory limitation. This study evaluates implementations of the three shuffling methods in our prototype in-memory MapReduce engine, which employs high performance interconnects such as InfiniBand and Intel Omni-Path. Our results suggest that DSA w/ SMS is the only viable solution for extremely skewed data distributions. We also present a detailed investigation of the performance of CSA and DSA in various skew situations.

key words: *MapReduce, Shuffle, Skew, libfabric, Intel Omni-Path*

1. Introduction

1.1 MapReduce for HPC

MapReduce [1] is the very foundation of numerous computing frameworks [2]–[9]. As suggested by the authors of [10], now there is a high demand for HPC frameworks capable of handling so-called *“extreme”* big data [10] generated by higher-resolution scientific simulations. MapReduce is one promising candidate for such a framework, and various designs for several supercomputers have been studied so far [2]–[6].

A typical MapReduce execution consists of three distinct phases: map, shuffle, and reduce, and the shuffle phase serves as a collective communication [11]. The communication pattern of shuffling is essentially equivalent to that of `MPI_Alltoallv` in the message passing interface (MPI) [12], which performs all-to-all data exchange among participating processes. The all-to-all collective communication pattern is essentially important in supercomputing, and the work in [13] is one attempt to improve its performance with a sophisticated algorithm.

1.2 The Skew Problem

MapReduce frameworks need to tackle a problem called *“skew”* - a load imbalance problem in which a large portion of the entire working set could be assigned to a small number of worker processes over the shuffle phase. The skew problem is rarely a concern for traditional scientific applications where problems could be load-balanced either by nature or with some clever portioning techniques for initial data. However, with emerging data-intensive applications, the skew problem is harder to resolve since the data distribution is often unknown statically, and is highly unpredictable. In MapReduce, the shuffle phase is a major cause of skewed distributions. The major MapReduce implementations [7], [8] try to resolve the skew problem by *“spilling”* overflowed data to local storage. However, typical supercomputers such as Oakforest-PACS [14] and K computer [15] only equip shared file systems, providing no local storage on computing nodes. Such an environment makes the *spill-to-storage* solution less appealing since file access is relatively expensive.

The existing MapReduce implementations can be divided into the following two groups:

HPC-Oriented [2]–[6]: These implementations use MPI as communication framework. `MPI_Alltoallv`, the all-to-all collective communication routine in MPI, is often employed to implement the shuffle phase. The actual shuffle communication, therefore, is concealed within the MPI library, making it difficult to address MapReduce-specific problems like skewed distributions. It should be noted that these implementations maximize the capabilities of high performance interconnects since major MPI implementations are finely optimized for such hardware environments.

Commodity-Oriented [7], [8]: These implementations, as described previously, assume that computing nodes possess local disks, and utilize the local disks as temporary storage for data overflows caused by high skew. An in-memory execution of shuffling on HPC clusters with no node-local storage, therefore, is out of their scope. Moreover, current implementations primarily target commodity cluster environments with 1 or 10 GbE networks.

Thus, to the best of our knowledge, it is still an unsolved question whether it is possible to address the skewed distribution problem of MapReduce shuffling in an *in-*

Manuscript received January 7, 2019.
Manuscript revised May 19, 2019.
Manuscript publicized July 29, 2019.

[†]The authors are with University of Tsukuba, Tsukuba-shi, 305–8577 Japan.
[††]The author is with the Faculty of Environment and Information Studies, Keio University, Fujisawa-shi, 252–0882 Japan.
a) E-mail: hd.11235813.revi1214@gmail.com

DOI: 10.1587/transinf.2019PAP0019

memory manner.

1.3 Proposal

In response to the above question, this study proposes and examines the following three shuffling methods, specifically designed to address the problem of skew in in-memory MapReduce engines:

Coupled Shuffle Architecture (CSA) shuffles both blocks (units of shuffle transfer) and meta-blocks (metadata of blocks) in a single pairwise all-to-all exchange.

Decoupled Shuffle Architecture (DSA) separates the shuffling of meta-blocks and blocks into two different all-to-all exchanges, referred to as *meta-shuffle* and *block-shuffle*, respectively. The meta-blocks are shuffled in a pairwise exchange, followed by the corresponding block-shuffle, which uses a naive all-to-all exchange. This design attempts to mitigate the performance degradation caused by *stragglers*.

DSA with Skew-Aware Meta-Shuffle (DSA w/ SMS) extends the meta-shuffle phase of the DSA method, and, in two consecutive meta-shuffles, autonomously determines the proper placement of the shuffling blocks based on the memory consumption of each process. With this design, shuffling can still be executed *in-memory*, even in an extremely skewed situation where some processes could exceed their memory capacity.

To evaluate these three methods, we implemented a prototype in-memory MapReduce engine that supports three network libraries, BSD socket, OFA verbs, and OFI libfabric, to make the most of high performance interconnect networks. Our codes are available on Bitbucket for reproducibility [16]. We evaluated our implementation of the three methods on 1024 nodes of the Oakforest-PACS supercomputer, showing that only the DSA w/ SMS method successfully executed in situations of extreme skew.

1.4 Organization

This paper is structured as follows: Section 2 explains conventional collective communication algorithms for shuffling, and their problems. Section 3 describes design details of the three proposed shuffling methods, CSA, DSA, and DSA w/ SMS, followed by their implementation details on our prototype in-memory MapReduce engine in Sect. 4. Section 5 reviews the results of the experiments we performed on the OFP cluster, as well as discussion about the performance and the skew tolerance of each method. Section 6 describes existing work on solving the skew problem, and, finally, concluding remarks are given in Sect. 7.

2. Conventional Collective Communication Algorithms for Shuffling

2.1 The Skew Problem

Skew is one important problem in the field of distributed computing, and generally indicates a state in which computational load notably varies among participating processes. In this paper, a *"skewed"* distribution specifically refers to a state in which a large portion of the entire working set is assigned to a small number of worker processes over the shuffle phase.

Under a skewed distribution, some highly loaded processes called *stragglers* could determine the execution time of the entire shuffling operation. In the worst case, the shuffling just fails to execute because of the *stragglers* consuming too much resource and exceeding the capacity of the computing nodes. Thus, resilience to a skewed distribution is an important aspect when designing collective communication algorithms for MapReduce shuffling.

2.2 The Naive Algorithm

In a naive algorithm, each process independently fetches data from the other processes. Since it requires no synchronization among processes, a naive algorithm is suitable for a highly skewed situation where the entire shuffling process could be affected by a few *stragglers*. Compared to other collective communication algorithms designed for MPI [17], the naive algorithm can be considered as the simplest algorithm to realize an all-to-all exchange. The implementation of `MPI_Alltoallv` in MPICH (v3.2.1) [18], one of the major implementations of MPI, consists of a series of asynchronous `MPI_Isend/Irecv` calls followed by an `MPI_Waitall` routine to wait for their completion. Thus, it can be classified as a naive method. Hadoop and Spark, the two major engines of MapReduce, employ a simple implementation of the naive algorithm in which each worker issues HTTP GET requests to the other workers. In contrast to these simpler implementations, the MPICH implementation optimizes the algorithm by limiting the number of concurrent send/recv operations and scattering the order of destinations among the ranks, which results in well-balanced load on the network across the system.

2.3 The Pairwise Algorithm

The pairwise algorithm is another all-to-all exchange algorithms designed for MPI. In a pairwise algorithm, processes pair and exchange data in a series of steps, and, at the end of each step, the two processes in each pair are synchronized. Thus, the algorithm is applicable only to a power-of-two number of processes. MPICH (v3.2.1) utilizes the pairwise algorithm to implement its `MPI_Alltoall` collective for long messages and power-of-two number of processes [19].

Figure 1 depicts the flow of a pairwise algorithm with four processes. At first, each process holds four data values (0, 1, 2, 3), and, going through the procedure, data sharing the same value are transferred to the same process. For n processes, the algorithm requires $n - 1$ steps to perform a complete all-to-all communication, and, in each step, $\frac{n}{2}$ connections are utilized. The generalized rule for pair for-

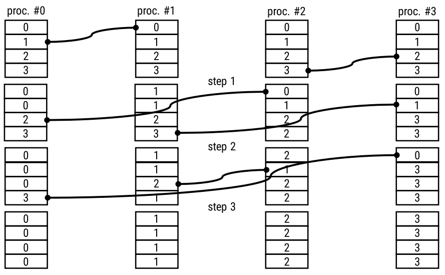


Fig. 1 Pairwise all-to-all exchange algorithm

mation is as follows: *In step k, a process with rank i forms a pair with the process (i XOR k).* Although the pairwise algorithm, as described above, is applicable only when there are power-of-two number of processes, the algorithm can be generalized for non-power-of-two number of processes as follows: *In step k, a process with rank i receives data from the process* $(i - k)$ mod n, *and sends data to the process* $(i + k)$ mod n.

With synchronization between consecutive steps, the pairwise algorithm ensures that any process communicates with at most one process at a time. Specifically, the maximum possible number of concurrent data transfers is $\frac{n}{2}$ in total, whereas, with the naive method, the number can be as large as $(n - 1) \times n$. This constraint enables more efficient utilization of network bandwidth. One disadvantage of the per-step synchronization, however, is that *stragglers* will affect the entire exchange process. Hence, the pairwise algorithm is considered to be only effective in low or zero skew situations.

In summary, although the pairwise algorithm is more sophisticated and achieves better network utilization over the naive algorithm, the algorithm is not optimal for a highly skewed condition with some *stragglers* affecting the entire shuffle execution.

3. Proposal

Based on the two algorithms described in the previous section, this study designs the following three all-to-all communication methods, CSA, DSA, and DSA w/ SMS, for MapReduce shuffling. Figure 2 shows communication flow of the three methods. First, the following two terms are defined to explain the methods in detail.
- *Block*: Unit of shuffle transfer
- *Meta-block*: The metadata of a corresponding block (Such as size, identifier, etc.)

Coupled Shuffle Architecture (CSA): In CSA, both meta-blocks and blocks are shuffled in a single pairwise all-to-all exchange. Following the pairwise algorithm, each process first exchanges meta-blocks with a peer. Next, the processes reference the size attribute given by the meta-block, allocate a receive buffer, and then exchange the blocks themselves. At the end of the step, each process synchronizes with its peer, limiting the number of concurrent transfers on each process to at most one. Thus, this method avoids overloading a certain network path. However, under a highly skewed situation, *stragglers* can impede the entire shuffling process. (Fig. 2 (a))

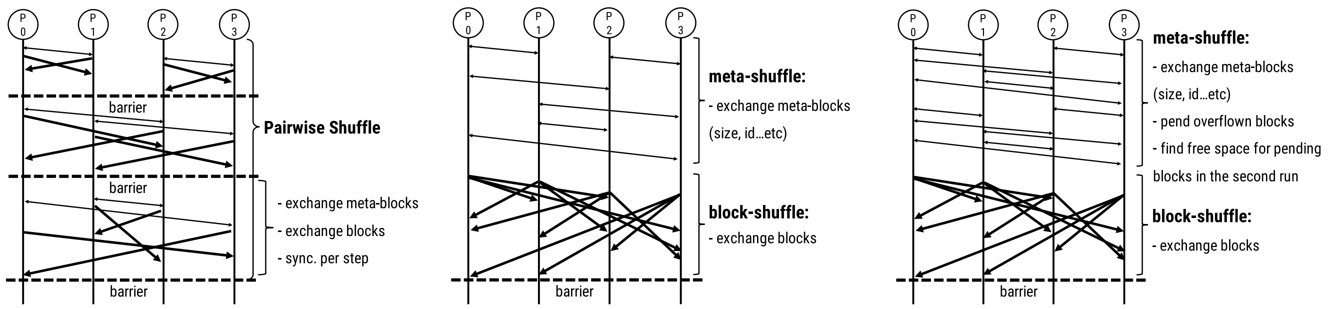
Decoupled Shuffle Architecture (DSA): DSA separates the shuffling of meta-blocks and blocks into two distinct all-to-all exchanges, referred to as *meta-shuffle* and *block-shuffle*, respectively. First, meta-blocks are shuffled in a pairwise exchange. Since the size of a meta-block is constant and independent of the size of the corresponding block, it is reasonable to assume that the meta-shuffle phase does not suffer from skewed distribution. As for the block-shuffle, blocks are shuffled in a naive all-to-all communication to relieve the impediment of *stragglers* in highly skewed situations. Note that, for a fair share of the network resource, a process chooses its communication peer in the same order as the pairwise-based meta-shuffle. Only a single global barrier operation is performed at the end of the block-shuffle phase to synchronize the worker processes. (Fig. 2 (b))

DSA with Skew-Aware Meta-Shuffle (DSA w/ SMS): DSA w/ SMS extends the meta-shuffle phase of the DSA method, and, in two consecutive meta-shuffles, autonomously determines the proper placement of the shuffling blocks based on the memory consumption of each process. (Fig. 2 (c))

Algorithm 1 and 2 explain the procedures of the DSA w/ SMS meta-shuffle on the receiver and the sender, respectively. The receiver first iterates through the array of meta-blocks received from the sender, and, for each meta-block, checks if the corresponding block can fit in its memory. If the sum of the size attribute given by the meta-block (*meta_blk.size*) and the current memory consumption (*cur_size*) is bigger than the maximum allowed size (*max_size*) of the node, the receiver stops accepting the blocks, and returns the number of blocks accepted to the sender. Note that the *max_size* parameter can easily be determined based on the memory capacity on each node. The sender, after receiving the number of blocks accepted by the receiver, marks rejected blocks as *"pending"* by inserting them into the *pending_blks* array. On the second run of the meta-shuffle, the sender tries to send the pending blocks to other receivers in a pairwise manner, delegating the blocks to processes which still have available memory. In other words, the overflowed blocks have been *"spilled"* to the remote memory.

After the second meta-shuffle, a block-shuffle is performed in the same way as in the original DSA method. Blocks are transferred to the process that has accepted the corresponding meta-block. After the block-shuffle, the user-defined reduce operation is performed, freeing memory to accept the blocks which once had been rejected due to lack of space. The workers then repeat the same series of the meta-shuffle and the block-shuffle until every block has been accepted by its primary owner. With the mechanism above, shuffling can still be performed *in-memory*, even in an extremely skewed situation where some processes could exceed their memory limitation.

Given the above observations, to enable DSA w/ SMS,



(a) Coupled Shuffle Architecture (CSA) (b) Decoupled Shuffle Architecture (DSA) (c) DSA with Skew-Aware Meta-Shuffle (DSA w/ SMS)

Fig. 2 Communication flow diagrams of CSA, DSA, and DSA w/ SMS

Algorithm 1 Receiver-side Pseudocode of SMS

```
$max\_size = USER\_DEFINED$
$cur\_size = 0$
function MetaShuffleSrv($peer\_rank$)
    recv_from($peer\_rank$, $meta\_blks$)
    $num\_accepted = 0$
    for $meta\_blk$ in $meta\_blks$ do
        if $cur\_size + meta\_blk.size > max\_size$ then
            break
        end if
        $cur\_size$ += $meta\_blk.size$
        $num\_accepted$++
    end for
    send_to($peer\_rank$, $num\_accepted$)
end function
```

Algorithm 2 Sender-side Pseudocode of SMS

```
$pending\_blks = []$
function MetaShuffleCli($peer\_rank$)
    $meta\_blks$ = get_meta_blks_for($peer\_rank$)
    send_to($peer\_rank$, $meta\_blks$)
    recv_from($peer\_rank$, $num\_accepted$)
    for $i = num\_accepted$ to $meta\_blks.length$ do
        $pending\_blks$.append($meta\_blks[i]$)
    end for
end function
```

the reduce operation must:
1) Be associative: $a \otimes (b \otimes c) = (a \otimes b) \otimes c$
2) Generate less data than input
Examples of such workloads include *group-by aggregate* tasks, such as word count, or low-selectivity relational join operations.

4. Implementation

This section describes the implementation of our prototype MapReduce engine specifically designed for evaluating the three shuffling methods explained in Sect. 2. The design of the engine is largely inspired by Resilient Distributed Datasets (RDD) [20] - the core API of Apache Spark. Our implementation, which is written in 6020 lines of C/C++ code, is publicly available on Bitbucket [16]. The following subsections give further explanation on two major internal modules, RDD and Shuffle Handler.

4.1 Resilient Distributed Datasets (RDD)

This module implements a subset of features provided by Spark RDD. Our prototype implementation provides three types of RDD: `KeyValueRDD` for holding ($key, value$) pairs, `KeyValuesRDD` for holding ($key, < value1, value2, \dots >$) pairs, and `DummyRDD` for generating artificial skewed shuffle communication. With DummyRDD, data of a user-specified size is first divided into 32-MiB partitions, and then distributed across worker processes. The distribution is controlled by another user-defined parameter called *skewness*. The skewed distribution of data is reproduced in the following manner:
1. Each RDD partition is divided into smaller chunks. The number of chunks per partition is equal to the total number of RDD partitions, and thus the size of each chunk is determined as follows:

$$chunk\_size = \frac{partition\_size}{num\_partitions}$$

2. Each chunk is transferred to a random RDD destination partition. The random number generator for determining the destination is weighted with the following function:

$$w(x) = \frac{1}{(x+1)^{\alpha}}$$
$$(x \in \mathcal{Z},\ 0 \le x < num\_partitions,\ \alpha \in \mathcal{R},\ \alpha \ge 0)$$

This weight function reflects Zipf's law, which states that the i-th most frequently appeared element can occupy as much as $\frac{1}{i}$ of the entire dataset. Zipf's law is known to hold for many real-world datasets, including the word frequency of English and the access frequency of WWW pages. Setting α to 0 in the weight function above signifies a zero skew distribution. Increasing α represents an increasingly skewed distribution of the data.

4.2 Shuffle Handler

Prior to the shuffle phase, each RDD partition is packed into a byte array, referred to as *"a block"*, using the `msgpack` library and put into the Shuffle Handler. After the shuffle completes, each RDD partition gets its own blocks from the local Shuffle Handler using its partition ID. The Shuffle Handler updates the total size of the blocks it manages every time a block is put into/get from the module. As for the inter-process communication, the Shuffle Handler module supports three network libraries, BSD socket, OFA verbs, and OFI libfabric [21]. An all-to-all connection is established at the initialization of the module.

For both CSA and DSA, meta-blocks are exchanged in a pairwise all-to-all communication implemented with a series of blocking send/recv operations. Each meta-block contains the size of its corresponding block. As for the block-shuffle of DSA, the socket implementation utilizes non-blocking send/recv operations with a `poll` event loop, while the verbs and the libfabric implementations mutually trigger one-sided read operations.

5. Evaluation

The purpose of this study is to design and examine skew-tolerant collective communications for MapReduce shuffling. To achieve this goal, we evaluated the three shuffling methods, CSA, DSA, and DSA w/ SMS, implemented in our prototype MapReduce engine described in Sect. 4.

All experiments were performed on the Oakforest-PACS (OFP) [14] cluster system operated by the Joint Center for Advanced High Performance Computing (JCAHPC). The configuration of the OFP cluster system is exhibited in Table 1. MCDRAM, the 16-GiB high bandwidth memory on the Xeon Phi (KNL) processor package, was configured to be used as Last Level Cache (LLC). The OFP is equipped with an Intel(R) Omni-Path interconnect network forming a full-bisection fat-tree topology. Omni-Path supports multiple software interfaces, including psm2, OFA verbs, and BSD socket. In the following experiments, we used psm2, which is the native interface of the Omni-Path interconnect, via OFI libfabric library.

Table 1 Cluster configuration (OFP)

# nodes	**2 ~ 1,024**
OS	CentOS Linux release 7.2.1511
Kernel	Linux 3.10.0-327.36.3.el7
libfabric (ofi)	1.5.3
CPU	Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz, 68 cores (4 HT)
Memory	DRAM: 96 GiB, MCDRAM: 16 GiB (used as LLC)
Interconnect	Intel(R) Omni-Path (100 Gbps), Full-bisection Fat-tree
Parallel File System	DDN Lustre
File Cache System	DDN IME14K

5.1 Experiment 1: Comparison of Skew Tolerance

5.1.1 Overview

This experiment compared the skew tolerance of the three shuffling methods, CSA, DSA, DSA w/ SMS, on 128 nodes of the OFP system using an artificial shuffling workload. The workload first assigns 1 GiB of data per process, 128 GiB in total, and then shuffles the data using a specified skewness parameter. The *skewness* parameter, which is denoted as α in the previous section, was increased from 0.0 to 3.0, in increments of 0.5, and re-examined as skewness increased from 2.5 to 3.0 in increments of 0.1. In the DSA w/ SMS method, each process was constrained to a maximum of 80 GiB of memory. The memory space available on each node was limited to 82 GiB by the job scheduler, and any jobs that reached the memory limit were terminated by the job scheduler. Each method was executed three times. The presented results average each set of three executions.

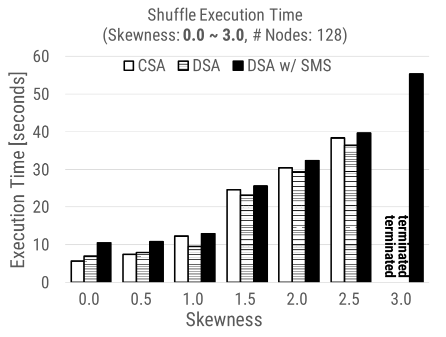
5.1.2 Results

Figure 3 (a) shows the execution times of the three shuffling methods as *skewness* increased from 0.0 to 3.0 in increments of 0.5. And Fig. 3 (b) shows the memory consumption of the most loaded process, the *straggler*, at each increment of skewness. When the skewness was 0.0, every process evenly received 1 GiB of blocks in total. As the skewness increased, the size of received blocks became progressively more varied across the processes, resulting in a sharp decline in the overall performance, as demonstrated in Fig. 3 (a). At a skewness of 3.0, the *straggler* received 90.69 GiB of blocks, in contrast to the initial, zero skew distribution of 1.00 GiB per process across all 128 nodes. Thus, on CSA and DSA, the entire shuffle processes were terminated by the job scheduler since the 82-GiB memory limit was breached on the *straggler* node. On the other hand, DSA w/ SMS successfully remained operational by exploiting remote memory, at the expense of execution time.

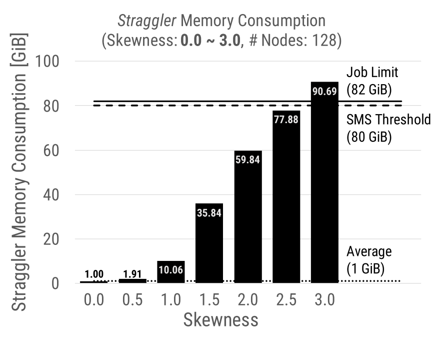
Throughout the experiment, the execution times of DSA w/ SMS were almost 3 seconds longer compared to that of the original DSA. This constant performance difference is reasonable because, as explained in Sect. 3, the DSA w/ SMS method requires additional operations during the meta-shuffle phase. Furthermore, the 3-second difference becomes less significant as increasing skewness dramatically extends execution time for all three methods.

Compared to DSA, CSA performed better under lower skew. At zero skew, CSA was 1.25 times faster than DSA. The all-to-all communication algorithms employed for shuffling blocks explain this performance difference. CSA, as described in Sect. 3, is impeded under high skew because stragglers hamper the synchronization that occurs at the end of each step of the pairwise algorithm. Conversely, under low skew, CSA balances the load on the network links, maximizing the efficient utilization of network bandwidth. This performance difference is further investigated in the subsection that follows.

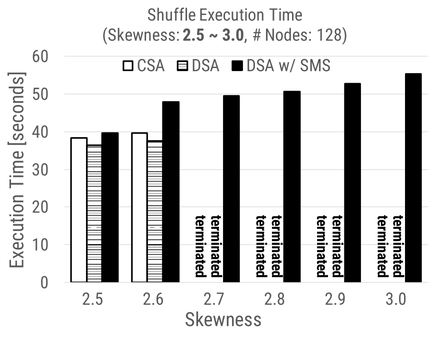
According to Fig. 3 (a), the memory consumption of the *straggler* process exceeded 82 GiB at some point in skewness between 2.5 to 3.0, causing the job scheduler to terminate the executions of CSA and DSA. To acquire



(a) Execution Times of Skewed Shuffling (Skewness: 0.0 ∼ 3.0)

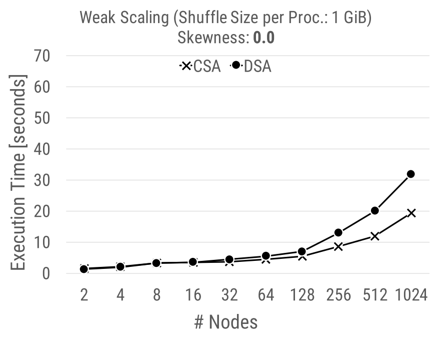


(b) Consumed Memory on *Straggler* Process
(Skewness: 0.0 ∼ 3.0)

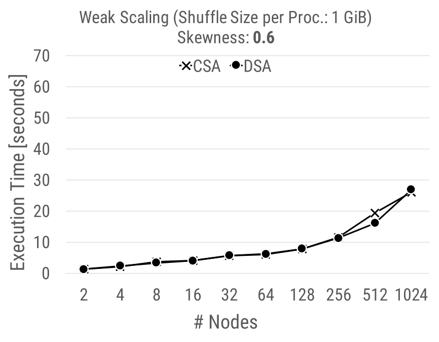


(c) Execution Times of Skewed Shuffling (Skewness: 2.5 ∼ 3.0)

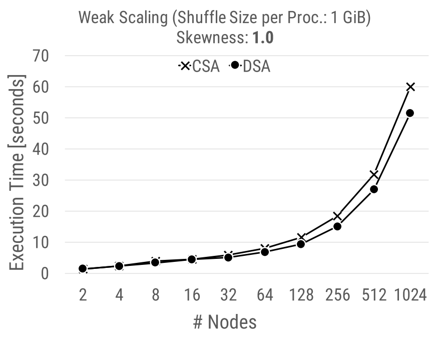
Fig. 3 Comparison of skew tolerance: CSA, DSA & DSA w/ SMS



(a) Skewness: 0.0



(b) Skewness: 0.6



(c) Skewness: 1.0

Fig. 4 Comparison of weak scaling performance: CSA & DSA

a more precise understanding of the behavior of shuffling methods around this skewness threshold, we examined the executions of the three shuffling methods under skewness values from 2.5 to 3.0, incremented in units of 0.1. Figure 3 (c) shows that, for the skewness equal to or greater than 2.7, CSA and DSA failed to operate. This occurred because the *straggler* consumed as much as 83.53 GiB of memory, surpassing the 82 GiB job limit.

As shown in Fig. 3 (c), when skewness increased from 2.5 to 2.6, DSA w/ SMS performance declined by a factor of 1.21, while CSA and the original DSA did not register significant changes in their execution times. This is reasonable since, at a skewness of 2.6, the memory consumption of the *straggler* reached 80.44 GiB. This just exceeded the 80-GiB-per-process threshold for DSA w/ SMS, automatically triggering the block delegation mechanism of the method. The additional block transfers over the network, which is unique to the DSA w/ SMS method, explain why only this method degraded its performance apparently when skewness reached 2.6. Note that, under this particular experimental settings, only the second round of shuffling was triggered by DSA w/ SMS. An even higher degree of skewness could lead to additional rounds of shuffling.

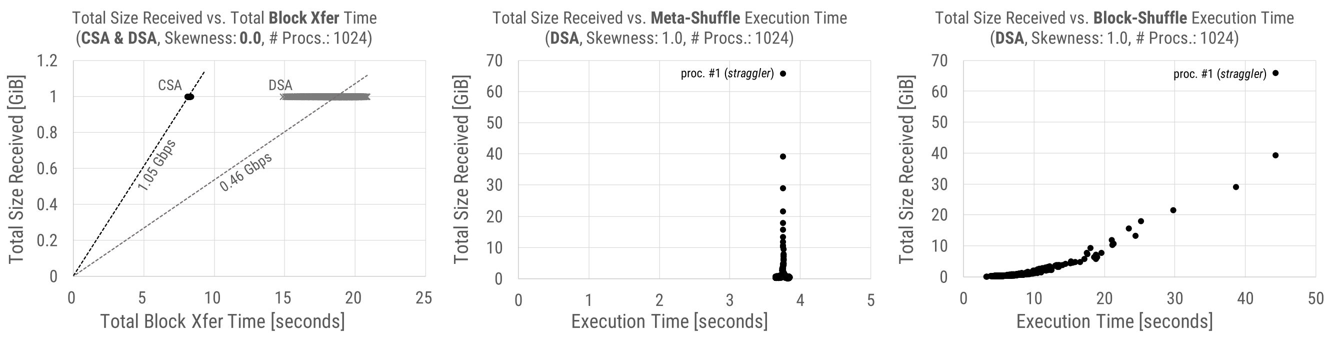
5.2 Experiment 2: Comparison of Weak Scaling Performance

5.2.1 Overview

This experiment further investigated the performance difference between CSA and DSA revealed in Experiment 1, by comparing their weak scaling performance. The artificial workload from Experiment 1 was also applied in this experiment, scaling the number of the worker processes from 2 to 1024. Only one worker process was launched per node, and 1 GiB of data was assigned to each process. The *skewness* parameter of the `DummyRDD` was increased from 0.0 to 1.0, in increments of 0.1. The selected results discussed below summarize those of the complete skewness range, detailing execution times for the selected skewness values of 0.0, 0.6, and 1.0. Note that, under all tested conditions, the memory consumption of the *straggler* process remained below the 82-GiB job limit enforced in this experiment.

5.2.2 Results

Figure 4 presents the execution times of CSA and DSA across increasing numbers of nodes at skewness values of 0.0, 0.6, and 1.0. Figure 4 (a) demonstrates that at zero skew,



(a) Block Transfer Time
(CSA & DSA, Skewness: 0.0)

(b) meta-shuffle Execution Time
(DSA, Skewness: 1.0)

(c) block-shuffle Execution Time
(DSA, Skewness: 1.0)

Fig. 5 Correlation between per process execution time and total block size

CSA outperformed DSA, especially as the number of nodes increased. At 1024 processes, in particular, CSA was 1.65 times faster than DSA. The standard deviation of the total size of blocks received per process was 0.00 GiB, since each process received 1.00 GiB of blocks under the zero-skew condition.

As the skewness increased, CSA performance declined under the influence of the *straggler*. Note that when the skewness reached 0.6, as shown in Fig. 4 (b), the performance ratio of the two methods was minimized, hovering at around 3%, regardless of the number of nodes. At 1024 processes, the standard deviation of the size of blocks received per process was 0.37 GiB, and the *straggler* received 6.25 GiB of blocks in total.

Under the most severe skew condition of this experiment, depicted in Fig. 4 (c), DSA was superior, achieving 1.16 times faster execution than CSA with 1024 processes. The standard deviation of the total size of blocks received per process was 2.90 GiB, and the *straggler* received as much as 65.86 GiB of blocks in total.

5.2.3 Advantage of CSA under Low Skew

Figure 4 indicates that, under low skew, CSA outperforms DSA. The result leads to the hypothesis that, with no *stragglers*, the pairwise all-to-all exchange employed in CSA can balance load on the network links, resulting in better utilization of the network bandwidth compared to DSA. To test this hypothesis, Fig. 5 (a) examines the correlation between block transfer time and block size received for the 1024-process result obtained under zero skew, as originally seen in Fig. 4 (a). Each plot point in Fig. 5 (a) compares the block transfer time and the total block size of the corresponding worker. The slope of the line connecting the origin and each point indicates the transfer bandwidth.

The 1024 plot points for CSA gather around the same temporal location, indicating that the block transfer times were almost identical among the processes. The average block transfer time was 8.19 seconds, and the standard deviation was 0.03 seconds. Since each process evenly received 1.00 GiB of blocks when the skewness was 0.0, the transfer bandwidth averaged 1.05 Gbps.

Contrarily, the plot points for DSA spread along the *x*-axis direction, indicating that the transfer time varied among the processes. The average block transfer time was 18.57 seconds, and the standard deviation was 1.21 seconds. Thus, the average transfer bandwidth was 0.46 Gbps, which was 2.28 times lower than that of CSA.

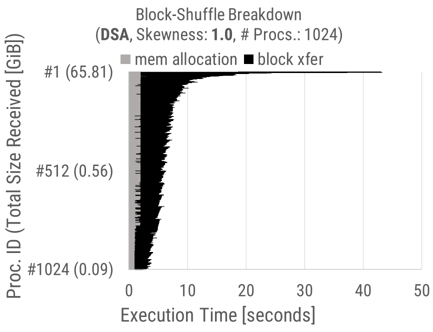
Summarizing the above results, we conclude that CSA, which uses the Pairwise exchange for the block shuffling, can effectively utilize the network bandwidth under low skew condition.

5.2.4 Performance Impact of *Straggler* under High Skew

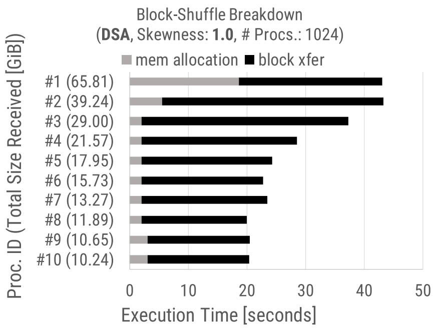
Figure 4 indicates that the performance of both CSA and DSA is affected by the *straggler*. As skewness increased from 0.0 to 1.0, CSA performance in the 1024-process scenario degraded by a factor of 3.11. Although considered to be more efficient than CSA in high skew situations, DSA performance also declined by a factor of 1.61. As the skewness increased, more and more blocks were transferred to the *straggler* process. At the maximum tested skewness of 1.0, 51.45% of the blocks were accumulated by the *straggler*. Based on this analysis, we can infer that the performance of the entire shuffle execution is constrained by the *straggler*.

To test the above hypothesis, for each phase of DSA shuffling, the meta-shuffle and the block-shuffle, we examined the correlation between process execution time and the total size of blocks received per process for the 1024-process scenario when the skewness was 1.0, as originally seen in Fig. 4 (c). Each plot point in Fig. 5 (b) and Fig. 5 (c) compares the execution time and the total block size of the corresponding worker for the meta-shuffle and block-shuffle, respectively.

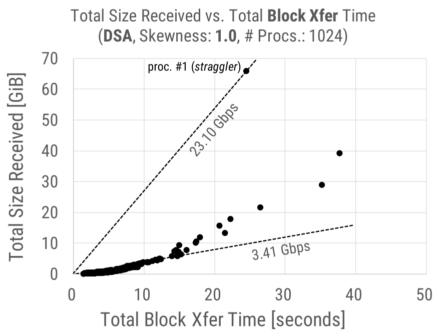
Figure 5 (b) indicates that the execution times of the meta-shuffle phase were fairly consistent, at around 3.70 seconds, regardless of the aggregated size of the blocks corresponding to the meta-blocks. This result is reasonable since, as described in Sect. 3, the meta-shuffle phase of DSA is implemented with the pairwise all-to-all exchange



(a) Breakdowns: 1024 Processes



(b) Breakdowns: Top 10 Processes



(c) Total Block Size vs. Block Transfer Time
straggler utilizes higher bandwidth than the others.

Fig. 6 Breakdowns of block-shuffle execution time: DSA (Skewness: 1.0)

in which the processes are synchronized per step, and meta-blocks are uniform in size, unlike, and independent of, the size of their corresponding blocks.

The block-shuffle phase depicted in Fig. 5 (c) reveals a strong positive correlation between the per process execution time and the total block size received. The execution time of the block-shuffle on the *straggler* process, which received 65.81 GiB of blocks in total,was 44.30 seconds, consuming 85.94% of the entire shuffle execution time of 51.55 seconds. Thus, we confirm that the performance of the entire shuffle execution is constrained by the *straggler*.

As can be seen in Fig. 5 (c), except for the *straggler*, the block-shuffle execution time is almost proportional to the total size of blocks received. To explore this exceptional behavior of the *straggler*, we examined the breakdown of the block-shuffle execution on each process. The two main process components of the block-shuffle are 1) allocation of receive buffers, and 2) transmission of blocks.

Figure 6 (a) breaks down the execution of 1024 processes, and Fig. 6 (b) highlights the top ten most loaded processes. In the figures, each process is assigned a unique identifier (#1 - #1024) in descending order of the total size of blocks it received. Overall, as observed in Fig. 6 (a), nearly 60% of the block-shuffle phase was spent transferring blocks, and that ratio increased as the process received more blocks. Interestingly, Fig. 6 (b) reports that process #1, the *straggler*, exhibited a shorter transfer time than that of the process #2.

To enhance our understanding of these phenomena, we examined the correlation between, for each of the 1024 processes, the time spent for transferring blocks and the total size of blocks received. The results in Fig. 6 (c) shows that the transfer bandwidth, which is indicated by the slope of the line connecting the origin and each point, increased as the process received more blocks. For most processes, the transfer bandwidth was approximately 3.41 Gbps, while the process # 1, the *straggler*, recorded a transfer bandwidth of 23.10 Gbps. These results make it apparent that the *straggler* can utilize higher bandwidth than the other process, spending relatively less time on the block-shuffle execution as shown in Fig. 6 (b).

6. Related Work

6.1 Remote Memory Utilization

SpongeFiles [22] implements a Java library that offers a spilling mechanism exploiting remote memory. SpongeFiles requires a dedicated server process to track computing node memory usage. A worker process attempting to perform remote spills must first request a list of nodes with available memory from the tracking server. In a large-scale environment, this could cause an excessively heavy load on the tracking server. To mitigate this issue, our study proposes a server-less method of remote spilling that is embedded in the all-to-all collective communication of shuffling.

The authors of Infiniswap [23] propose an OS-level remote memory paging system, which utilizes RDMA technologies. Since it is implemented at the OS layer, any application running on top of an Infiniswap-enabled system could transparently utilize remote memory without modification. While convenient in one respect, this generalized functionality comes at the cost of difficulties optimizing the system for any specific application. For example, the map output of MapReduce that will be consumed by a local reducer should not be spilled to remote memory. Furthermore, since Infiniswap exploits the swapping mechanism of the host OS, it must provide fault tolerance for the spilled data. To satisfy this requirement, Infiniswap must asynchronously replicate the spilled data to a local disk as well. For a shared-nothing computing model like MapReduce, however, fault tolerance is not necessarily a *must-have*, since the lost portion of a working set could be recomputed. Our study aims to design a MapReduce-specific spilling mechanism at the framework level.

To redistribute data during the shuffle phase, most MapReduce implementations by default use simple hash partitioning, which often ignores the distribution characteristics of keys and thus triggers undesired skewed distribution. Both Hadoop and Spark allow users to write custom partitioners and optimize distribution based on the characteristics of their workloads. However, as emphasized by the authors of [1] and [22], MapReduce was originally designed to provide an easy-to-use parallel programming model for non-expert programmers with little knowledge of distributed systems. Hence, various sophisticated partitioning strategies have been proposed to *transparently* address the skew problem [24]–[26].

6.2 Communication Optimization

One commonly used technique for optimizing shuffle performance, which also is leveraged by Hadoop, is to overlap the shuffle phase with computation phases, that is, map or reduce, and hide the expensive communication latency of shuffling. However, as stated by the authors of [27], this approach could lead to inefficient memory utilization due to buffering of the shuffled data, especially given the recent trend of decreasing memory-to-core ratio. For an in-memory MapReduce engine where fetched data must reside in memory, this design is less favorable.

Another approach to optimize existing MapReduce frameworks for the HPC environment is to utilize high performance interconnects. The RDMA technology has been proven to be effective for accelerating the shuffle communication by several studies [28]–[30].

All the studies described above focus on the optimization of Apache Hadoop or Spark, the two most popular MapReduce implementations, which by nature is designed for commodity cluster systems. Thus, these studies still employ some types of storage system for placing intermediate data during the shuffle phase. Contrarily, our study designs and implements an *in-memory* MapReduce engine that exploits remote memory via high performance interconnect, completely eliminating the expensive file I/Os from the shuffle execution path.

6.3 Application-Level Solutions for Skewed Data

While the focus of this paper and the work described above is on addressing the skew problem at the framework level, offering an application-specific remedy could be another effective way to mitigate the skewness. The work in CloudRAMSort [31] and SDS-Sort [32] are two examples of addressing the skewed data in the context of distributed sorting. The idea is to utilize knowledge about the data itself by partitioning the keyspace based on a result of random-sampling before the data is passed to a lower-level framework like MPI. Our solution, on the other hand, does not rely on any characteristics of the raw data, but only on the size of serialized byte arrays. Thus it is a more generic solution for the problem, and intends to reduce the burden on the application developers.

7. Conclusions

In this paper, we proposed and examined the three in-memory shuffling methods, CSA, DSA, and DSA w/ SMS, designed to address the problem of skew in MapReduce shuffling. CSA shuffles both blocks and meta-blocks in a single pairwise all-to-all exchange, which is particularly well-suited to a low skew situation. The second method, DSA, separates the shuffling of meta-blocks and blocks, applying two different types of all-to-all exchanges. The DSA approach aims to mitigate the impact of *stragglers* in highly skewed distributions. The final, and most complex method, DSA w/ SMS, autonomously determines the proper placement of the shuffling blocks based on the memory consumption of each process. DSA w/ SMS targets extremely skewed conditions where some processes could exceed their memory limit. To evaluate these three methods, we implemented a prototype in-memory MapReduce engine that supports three network libraries, BSD socket, OFA verbs, and OFI libfabric, to make the most of high performance interconnect networks.

Through a series of experiments, we examined the performance and the skew tolerance of the three shuffling methods. Our results confirmed that only the DSA w/ SMS method, which exploits remote memory via high performance interconnect, successfully executes in situations of extreme skew. Regarding weak scaling performance, CSA outperformed DSA by a factor of 1.65 under low skew conditions. This was possible because CSA achieves more efficient network utilization due to the application of the pairwise exchange.

Acknowledgments

This work is partially supported by JST CREST Grant Numbers JPMJCR1303 and JPMJCR1414, JSPS KAKENHI Grant Number JP17H01748, Oakforest-PACS at the CCS, University of Tsukuba and project commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Commun. ACM, vol.51, no.1, pp.107–113, Jan. 2008.
[2] T. Hoefler, A. Lumsdaine, and J. Dongarra, "Towards Efficient MapReduce Using MPI," Recent Advances in Parallel Virtual Machine and Message Passing Interface, vol.5759, pp.240–249, Springer Berlin Heidelberg, 2009.
[3] H. Mohamed and S. Marchand-Maillet, "MRO-MPI: MapReduce overlapping using MPI and an optimized data exchange policy," Parallel. Comput., vol.39, no.12, pp.851–866, 2013.
[4] M. Matsuda, N. Maruyama, and S. Takizawa, "K MapReduce: A scalable tool for data-processing and search/ensemble applications on large-scale supercomputers," 2013 IEEE International Conference on Cluster Computing (CLUSTER), pp.1–8, 2013.
[5] Y. Guo, W. Bland, P. Balaji, and X. Zhou, "Fault Tolerant MapReduce-MPI for HPC Clusters," Proc. International Conference for High Performance Computing, Networking, Storage and Analysis, SC '15, New York, NY, USA, pp.34:1–34:12, ACM, 2015.
[6] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Mimir: Memory-Efficient and Scalable MapReduce for

Large Supercomputing Systems," 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp.1098–1108, May 2017.
[7] Apache Hadoop, http://hadoop.apache.org/.
[8] Apache Spark, http://spark.apache.org/.
[9] In-Memory MapReduce - Apache Ignite, https://ignite.apache.org/features/mapreduce.html.
[10] S. Matsuoka, H. Sato, O. Tatebe, M. Koibuchi, I. Fujiwara, S. Suzuki, M. Kakuta, T. Ishida, Y. Akiyama, T. Suzumura, K. Ueno, H. Kanezashi, and T. Miyoshi, "Extreme Big Data (EBD): Next Generation Big Data Infrastructure Technologies Towards Yottabyte/Year," Supercomputing Frontiers and Innovations, vol.1, no.2, 2014.
[11] Collective Communication, https://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node63.html.
[12] The standardization forum for the Message Passing Interface (MPI), http://mpi-forum.org.
[13] C. Xu, M.G. Venkata, R.L. Graham, Y. Wang, Z. Liu, and W. Yu, "SLOAVx: Scalable LOgarithmic AlltoallV Algorithm for Hierarchical Multicore Systems," 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pp.369–376, May 2013.
[14] About Oakforest-PACS, http://jcahpc.jp/eng/ofp_intro.html.
[15] What is K?, http://www.aics.riken.jp/en/k-computer/about/.
[16] H. Daikoku, Source code of the prototype MapReduce implementation, https://bitbucket.org/hdaikoku/pmr/src/devel/.
[17] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of Collective Communication Operations in MPICH," The International Journal of High Performance Computing Applications, vol.19, no.1, pp.49–66, 2005.
[18] Implementation of MPI_Alltoallv in mpich-3.2.1, https://github.com/pmodels/mpich/blob/v3.2.1/src/mpi/coll/alltoallv.c.
[19] Implementation of MPI_Alltoall in mpich-3.2.1, https://github.com/pmodels/mpich/blob/v3.2.1/src/mpi/coll/alltoall.c.
[20] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," Proc. 9th USENIX Conference on Networked Systems Design and Implementation, NSDI '12, Berkeley, CA, USA, pp.15–28, USENIX Association, 2012.
[21] Libfabric, https://ofiwg.github.io/libfabric/.
[22] K. Elmeleegy, C. Olston, and B. Reed, "SpongeFiles: Mitigating Data Skew in Mapreduce Using Distributed Memory," Proc. 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, New York, NY, USA, pp.551–562, ACM, 2014.
[23] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K.G. Shin, "Efficient memory disaggregation with Infiniswap," 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), pp.649–667, USENIX Association, March 2017.
[24] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "SkewTune: Mitigating skew in Mapreduce applications," Proc. 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, New York, NY, USA, pp.25–36, ACM, 2012.
[25] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu, "Handling partitioning skew in MapReduce using LEEN," Peer-to-Peer Networking and Applications, vol.6, no.4, pp.409–424, Dec. 2013.
[26] Y. Gao, Y. Zhou, B. Zhou, L. Shi, and J. Zhang, "Handling Data Skew in MapReduce Cluster by Using Partition Tuning," Journal of Healthcare Engineering, vol.2017, pp.1–12, 2017.
[27] B. Nicolae, C.H.A. Costa, C. Misale, K. Katrinis, and Y. Park, "Leveraging Adaptive I/O to Optimize Collective Data Shuffling Patterns for Big Data Analytics," IEEE Trans. Parallel Distrib. Syst., vol.28, no.6, pp.1663–1674, 2017.
[28] M. Wasi-ur-Rahman, N.S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D.K. Panda, "High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand," 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, pp.1908–1917, 2013.
[29] X. Lu, D. Shankar, S. Gugnani, and D.K. Panda, "High-performance design of apache spark with RDMA and its benefits on various workloads," 2016 IEEE International Conference on Big Data (Big Data), pp.253–262, Dec. 2016.
[30] H. Daikoku, H. Kawashima, and O. Tatebe, "On Exploring Efficient Shuffle Design for In-memory MapReduce," Proc. 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR '16, pp.6:1–6:10, 2016.
[31] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani, "CloudRAMSort: Fast and Efficient Large-scale Distributed RAM Sort on Shared-nothing Cluster," Proc. 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, pp.841–850, 2012.
[32] B. Dong, S. Byna, and K. Wu, "SDS-Sort: Scalable Dynamic Skew-aware Parallel Sorting," Proc. 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC '16, pp.57–68, 2016.
[33] IME®FLASH-NATIVE DATA CACHE — DDN, https://www.ddn.com/products/ime-flash-native-data-cache/.

Appendix: Skew Dealing Method with Shared File System

As described in Sect. 1, this study assumes a target environment of typical HPC clusters such as Oakforest-PACS (OFP) and K computer, which have no local disks on computing nodes. Since accessing files on shared storages, such as Lustre or GlusterFS, is relatively expensive, such an environment makes the *spill-to-storage* solution employed by Hadoop and Spark less appealing. This section presents the results of the performance comparison between the DSA w/ SMS method and the so-called *On-Disk* method.

Figure A· 1 gives an overview of block transfer in the On-Disk method. In the meta-shuffle phase, the On-Disk method checks if the blocks can fit in node memory in the same way as the DSA w/ SMS method. Unlike the DSA w/ SMS method, the On-Disk method spills overflowed blocks over into the shared file system, making the blocks globally accessible to the remote processes. In the block-shuffle phase, block transfers over the network, as well as the shared file system, are performed. After the shuffling, each reduce task tries to fetch its own blocks from the local memory or the shared file system via the Shuffle Handler module.

We implemented the On-Disk method in our prototype MapReduce engine described in Sect. 4. We again evaluated its performance evaluation on 128 computing nodes of the OFP. There are two file systems available on the OFP: the HDD-based Lustre file system, and the SSD-based burst buffer, also known as *IME* [33]. The IME can be used for caching files on the Lustre, as well as for storing tempo-

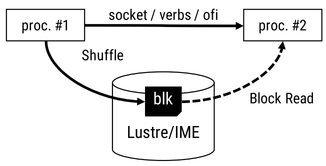
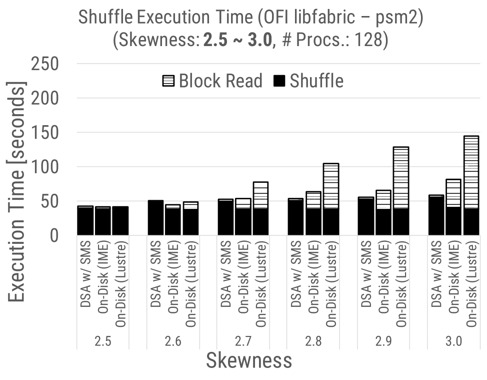
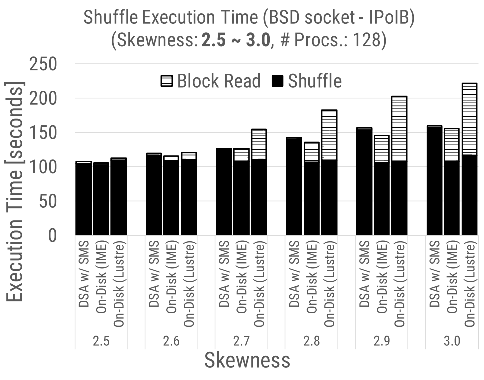


Fig. A· 1 Block transfer via shared file system



(a) OFI libfabric (psm2)



(b) BSD socket (IPoIB)

Fig. A·2 Comparison of skew tolerance: DSA w/ SMS & On-Disk

rary files during job execution. For this experiment, the temporary directory for block spilling was set to either the Lustre or the IME, and we measured the shuffle execution time for each scenario. We also investigated the effect of network performance by executing the workload using two different network libraries, BSD socket and OFI libfabric, enabled respectively. The former library utilizes the IPoIB network of Intel Omni-Path, while the latter library is expected to demonstrate better performance because it leverages Omni-Path's native psm2 interface. All other parameters precisely replicated the conditions of the experiments detailed in Sect. 5.

Figure A·2 shows the execution times of the two shuffling methods, DSA w/ SMS and On-Disk, as the *skewness* parameter was increased from 2.5 to 3.0, in increments of 0.1. The *Shuffle* segments of each bar represent the overall shuffle execution time, while the *Block Read* segments represent the amount of time that reduce tasks spent fetching blocks from the Shuffle Handler module and deallocating their memory regions. For the On-Disk method, the *Shuffle* segment includes the time spent spilling blocks over into the shared file system, and the *Block Read* segment includes the time spent reading the blocks from the shared file system.

The results, displayed in Fig. A·2 (a), suggest that, when using OFI libfabric, the advantage of in-memory shuffling in DSA w/ SMS becomes more meaningful as skewness increases. When the skewness was 3.0, the DSA w/ SMS improved the shuffling performance over the On-Disk (Lustre) and the On-Disk (IME) by factors of 2.49 and 1.39, respectively. Furthermore, the impact of expensive file I/Os becomes more apparent in the *Block Read* phase. In the *Shuffle* phase, multiple processes were spilling blocks to the file system in parallel, whereas, in the *Block Read* phase, only the single *straggler* was reading the blocks, resulting in the relatively longer execution of the phase. Although the IME improved the *Block Read* execution time by a factor of up to 3.23, the DSA w/ SMS method still achieved the fastest overall execution times in higher skew situations.

When using BSD socket, displayed in Fig. A·2 (b), the DSA w/ SMS method was no longer the best overall solution. For example, when the skewness was 2.9, the On-Disk (IME) was 7.20% faster than the DSA w/ SMS. The relatively poor performance of the IPoIB network is the most likely cause of the observed performance degradation.

Based on our analysis of the above results, we conclude that DSA w/ SMS is not a promising solution for the skew problem unless applied to systems equipped with high performance interconnects.

Harunobu Daikoku received M.Eng. from University of Tsukuba, Japan in 2018. His research area is high-performance computing and distributed system software.

Hideyuki Kawashima received Ph.D. from Science for Open and Environmental Systems Graduate School of Keio University, Japan. He was a research associate at Department of Science and Engineering, Keio University from 2005 to 2007. From 2007 to 2011, he was an assistant professor at both Graduate School of Systems and Information Engineering and Center for Computational Sciences, University of Tsukuba, Japan. From 2016 to 2018, he was an associate professor at Center for Computational Sciences, University of Tsukuba. From 2018, he is an associate professor at in Environment and Information Studies, Keio University.

Osamu Tatebe received a Ph.D. in computer science (1997, Univ. of Tokyo). He worked at Electrotechnical Laboratory (ETL), and National Institute of Advanced Industrial Science and Technology (AIST) until 2006. He is now a professor in Center for Computational Sciences at University of Tsukuba. His research area is high-performance computing, data-intensive computing, and parallel and distributed system software.