Memory-Efficient and Skew-Tolerant MapReduce Over MPI for Supercomputing Systems

Tao Gao, *Student Member, IEEE*, Yanfei Guo, *Member, IEEE*, Boyu Zhang, Pietro Cicotti, *Member, IEEE*, Yutong Lu, *Member, IEEE*, Pavan Balaji, *Senior Member, IEEE*, and Michela Taufer, *Senior Member, IEEE*

Abstract—Data analytics has become an integral part of large-scale scientific computing. Among various data analytics frameworks, MapReduce has gained the most traction. Although some efforts have been made to enable efficient MapReduce for supercomputing systems, they are often limited to fairly homogeneous workloads where equal partitioning of input data across tasks results in essentially equal output or temporary data generated on each task. For workloads that are more skewed, however, current implementations can result in imbalance in memory usage and, consequently, can cause a slowdown in execution time and a loss in data scalability. To tackle this problem, we enhance a previously published memory-conscious MapReduce over MPI framework called Mimir. Our enhancements to Mimir include combiner and dynamic repartition optimizations to minimize and balance memory usage and to achieve close to optimal balance of the memory usage across processes and to reduce the execution time by up to 12 times. Experimental results show that Mimir can scale to at least 3072 processes on the Tianhe-2 supercomputer on skewed datasets.

Index Terms—Skew mitigation, load balancing, high-performance computing, data analytics, MapReduce, memory efficiency, performance and scalability

1 Introduction

With the growth of simulation and scientific data, data analytics and data-intensive workloads have become an integral part of large-scale scientific computing. MapReduce [11] is one of the most popular programming models within the broad data analytics domain.

Most implementations of MapReduce, such as Hadoop [1] and Spark [37], target Linux-based commodity clusters whose features are significantly different from supercomputers in terms of operating systems, networks, and storage features. First, most large supercomputer installations do not provide on-node persistent storage (although this situation might change with chip-integrated NVRAM). Instead, storage is decoupled into a separate globally accessible parallel file system. Second, network architectures on many of the fastest machines in the world are proprietary. Thus, commodity-network-oriented protocols, such as TCP/IP or RDMA over Ethernet, do not work well (or work at all) on many of these networks. Third, system software stacks on these platforms, including the operating system and computational libraries, are specialized for scientific computing. For example, supercomputers such as the IBM Blue Gene/Q [2] use specialized lightweight operating systems that do not provide the same capabilities as those that a traditional operating system such as Linux or Windows might.

Some efforts have been made to enable efficient MapReduce implementations for supercomputers. These efforts can be divided into two categories: (1) tuning and deployment of popular MapReduce frameworks on high-performance computers [8], [23], [29], [36] and (2) design and building of new implementations of MapReduce on top of MPI (e.g., MapReduce-MPI, or MR-MPI [27] and Mimir [15]). Implementations of MapReduce over MPI have gained the most traction for two reasons: (1) they provide C/C++ interfaces that are more convenient to integrate with existing scientific applications compared with Java, Scala, or Python interfaces; and (2) they can make use of the high-speed interconnection network through MPI.

Independently from the platform on which they are executed, MapReduce jobs constantly flush intermediate data to node-local storage and read data back as jobs progress when there is load imbalance and the data cannot be held in the memory. This is an effective solution for traditional cloud environments because it is independent of how fast the

- *T. Gao is with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996, and also with the National University of Defense Technology, Changsha 410073, China. E-mail: tao.gao.nudt@hotmail.com.*
- *Y. Guo and P. Balaji are with the Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL 60439. E-mail: {yguo, balaji}@anl.gov.*
- *B. Zhang and M. Taufer are with the Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996. E-mail: zhang.boyu84@gmail.com, taufer@utk.edu.*
- *P. Cicotti is with NVIDIA, San Diego, CA 95051. E-mail: pcicotti@sdsc.edu.*
- *Y. Lu is with the National Supercomputing Center in Guangzhou, China, and also with the Sun Yat-sen University, Guangzhou 510275, China. E-mail: yutong.lu@nscc-gz.cn.*

Manuscript received 9 Jan. 2018; revised 7 May 2019; accepted 16 July 2019. Date of publication 28 May 2020; date of current version 29 June 2020. (Corresponding author: Michela Taufer.) Recommended for acceptance by A. Iamnitchi. Digital Object Identifier no. 10.1109/TPDS.2019.2932066

network is, what the overall memory capacity of the distributed system is, and what load imbalance issues other processes are facing. However, when executed on supercomputers which usually lack on-node persistent storage, these jobs may suffer from degrading performance because of the extensive I/O operations to the shared storage file system.

To tackle the problems associated with the loss of performance due to memory inefficiency and data skewness, we enhance a previously published memory-conscious MapReduce over MPI framework called Mimir [15]. The goal of Mimir is to efficiently process large datasets in memory. In doing so, however, it might sacrifice a small amount of performance for small datasets compared with MR-MPI. With Mimir, we have taken a significant first step to mitigate performance lost through in-memory processing. Mimir includes memory-efficient intermediary combiner operations, data skew strategies based on dynamic repartitions, and superkey split strategies. The results demonstrate that our approach not only reduces the memory usage but also significantly balances the memory usage for skewed datasets.

The contributions of this paper are twofold.

1) We design a memory-efficient and skew-tolerant MapReduce over MPI framework for supercomputing systems. Our design builds on top of Mimir's in-memory workflow and includes (a) a pipeline combiner workflow to use memory more efficiently and balance memory usage; (b) a new dynamic repartition method that mitigates data skew on MapReduce applications without obviously increasing their peak memory usage; and (c) a strategy for splitting single superkeys across processes and further mitigating the impact of data skew, by relaxing the MapReduce model constraints on key partitioning.
2) We evaluate the results of Mimir's in-memory workflow, the combiner workflow, the dynamic repartition workflow, and the superkey and splitting approach with respect to memory usage and performance (i.e., execution time) on the Tianhe-2 supercomputer for three benchmarks (i.e., word count, octree clustering, and join) and three different types of datasets: balanced data, value imbalanced data (i.e., the imbalance is reflected in the value distributions), and key-mapping imbalanced data (i.e., the imbalance is reflected in the unique key distribution across processes).

The rest of this paper is organized as follows: Section 2 discusses the status and drawback of the existing MapReduce implementations in supercomputers; Section 3 summarizes the original design of Mimir, which works well for balanced datasets but not for skewed datasets; Section 4 presents the design of the combiner optimization that helps with data skew in applications exhibiting associative and commutative properties such as word count; Section 5 presents the design of repartitioning methods that are more complex data-driven optimizations and can help with data skew in applications such as the octree clustering and join; Section 6 explains the integration of the proposed optimizations in Mimir; Section 7 presents the evaluation and analysis of the proposed optimizations; Section 8 discusses the related work; and Section 9 briefly summarizes our conclusions.

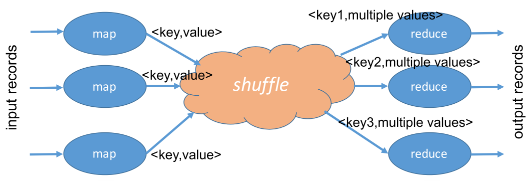


Fig. 1. *Map*, *shuffle*, and *reduce* phases of the MapReduce programming model.

2 Background

In this section, we review the MapReduce programming model and present MR-MPI's workflow. MR-MPI is the MapReduce over MPI framework that more closely matches the Mimir implementation, and thus we will use it for some of our performance comparisons.

2.1 MapReduce Programming Model

MapReduce is a programming model intended for data-intensive applications [11] that has proved suitable for a wide variety of applications. A MapReduce job usually involves three phases—*map*, *shuffle*, and *reduce*—as shown in Fig. 1. The *map* phase processes the input data using a user-defined map callback function and generates intermediate $\langle key, value \rangle$ (KV) pairs. The *shuffle* phase performs an all-to-all shuffle communication that distributes the intermediate KVs across all processes. In this phase, KVs with the same key are also merged and stored in $\langle key, multiplevalues \rangle$ (KMV) lists. The *reduce* phase processes the KMV lists with a user-defined reduce callback function and generates the final output.

2.2 MapReduce-MPI (MR-MPI)

MR-MPI supports the logical *map-shuffle-reduce* workflow in four phases: `map`, `aggregate`, `convert`, and `reduce`. The `map` and `reduce` phases are implemented by using user callback functions. The `aggregate` and `convert` phases are fully implemented within MR-MPI but need to be explicitly invoked by the user. The `aggregate` phase handles the all-to-all movement of data between processes. In the `aggregate` phase, MR-MPI calculates the data and buffer sizes and exchanges the intermediate KV pairs using `MPI_Alltoallv`. After the exchange, the `convert` phase merges all received KV pairs based on their keys.

Similar to traditional MapReduce frameworks, MR-MPI uses a global barrier to synchronize at the end of each phase. Because of this barrier, the job must hold all the intermediate data either in memory or on the I/O subsystem until all processes have finished the current stage. For large MapReduce jobs, intermediate data can use considerable memory. Especially for iterative MapReduce jobs where the same dataset is repeatedly processed, buffers for intermediate data need to be repeatedly allocated and freed. To avoid memory fragmentation, MR-MPI uses a fixed-size buffer structure called *page* to store the intermediate data. The coarse-grained memory allocation leads to efficiency problems: not all the allocated *pages* are fully utilized.

3 Leveraging In-Memory Storage

The first design goal of Mimir is to allow for a memory-efficient MapReduce implementation over MPI that ultimately

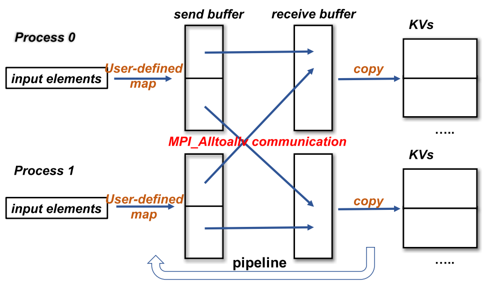


Fig. 2. Map workflow in Mimir with its map phase including both computation and shuffle communication stages.

allows users to run significantly large-sized data analytics in memory. To this end, Mimir introduces new objects and deploys pipelines and interleaves of computation and shuffle communication within the map phase to minimize unnecessary memory usage.

3.1 Intermediate Data Management

As in MR-MPI [25], Mimir builds on the concepts of KVs and KMVs. Moreover, it introduces two new objects, called KV containers (KVCs) and KMV containers (KMVCs), to help manage KVs and KMVs efficiently. A KVC is an opaque object that internally manages a collection of KVs in one or more buffer *pages* based on the number and sizes of the KVs inserted. A KVC provides read/write interfaces that Mimir can use to access the corresponding data buffer. The KVC tracks the use of each data buffer and controls memory allocation and deallocation. In order to avoid memory fragmentation, the data buffers are always allocated in fixed-size units whose size is configurable by the user. When KVs are inserted into the KVC, it gradually allocates more memory to store the data. When the data is read (consumed), the KVC frees buffers that are no longer needed. KMVCs are functionally identical to KVCs but manage KMVs instead of KVs.

3.2 Map and Reduce Phases

As in Hadoop [1] and Spark [37], the user of Mimir's basic workflow defines the `map` and `reduce` operations, which MPI processes then execute. Different from the master-worker architecture used in Hadoop[1] and Spark [37], however, Mimir is designed to be decentralized in order to increase scalability.

In the map phase shown in Fig. 2, each MPI process has a send buffer and a receive buffer. The send buffer is divided into p equal-sized partitions, where p is the number of processes executing a given MapReduce application. In other words, each partition corresponds to one process. The execution of the map phase starts with the computation stage. In this stage, the input data is transformed into KVs by the user-defined map function executed by each process. The new KVs are inserted into one of the send buffer partitions so that KVs with the same key are sent to the same process. The default partitioning method is based on the hash value of the key. Users can provide alternative partition algorithms that better suit their needs, but the overall workflow remains the same. If a partition in the send buffer is full, Mimir temporarily suspends the computation stage and switches to the shuffle communication stage. In this stage, all processes exchange their accumulated intermediate KVs using `MPI_Alltoallv`: each process sends the data in its send buffer partitions to the corresponding destination processes and receives data from all other processes in its receive buffer partitions. Once the KVs are in the receive buffer, each process moves the KVs into a KVC. The KVC serves as an intermediate holding area between the map and reduce phases. After the data has been moved to this KVC, the shuffle communication stage completes, and the suspended computation phase resumes. In this way, the computation and shuffle communication stages are interleaved, allowing them to process large volumes of input data without correspondingly increasing the memory usage.

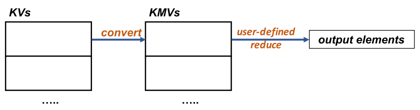


Fig. 3. Workflow of the reduce phase in Mimir with the two-pass algorithm to perform the KV-KMV conversion in memory.

In the reduce phase shown in Fig. 3, the input KVs are stored in a KVC that is generated by the map phase. The reduce phase starts with the conversion of KVs to KMV lists. Mimir adopts a two-pass algorithm to perform the conversion in memory. In the first pass, the size of the KVs for each unique key is gathered in a hash bucket and used to calculate the position of each KMV in the KMV container (KMVC), described in Section 3.1. In the second pass, the KVs are converted into KMV lists by inserting them into the corresponding position in the KMVC. When all the KVs are converted to KMV lists, the conversion is complete. Mimir then calls the user-defined reduce callback function on the KMVs.

3.3 Memory Usage Analysis

In the following, we assume that the lengths of each key and each value are the same. The Mimir software itself makes no such assumption; we use it here to simplify our memory usage analysis. Let us use n to represent the number of KVs assigned to one process during the shuffle communication and u to represent the number of unique keys in those KVs. We note that u is always equal to or smaller than n. In each process, only the memory usage of the intermediate data buffers (i.e., KVCs and KMVCs) depends on the dataset size. The spatial complexity of a given KVC is $O(n)$ and that of a given KMVC is $O(u) + O(n)$. If u is much smaller than or comparable to n, then the spatial complexity of the KMVC is $O(n)$, the same as that of KVC. Because the spatial complexity of KVC and KMVC is linearly dependent on the number of KVs partitioned to them, the memory usage is unbalanced if the intermediate KVs are not partitioned evenly.

4 Function-Driven Optimizations

In a large number of MapReduce applications the reduction function is both associative and commutative. From an implementation point of view these MapReduce applications support merging KV pairs before the reduce function. The merging process is called `combiner` optimization. In this section, we present combiner optimizations to minimize and balance the memory usage in Mimir.

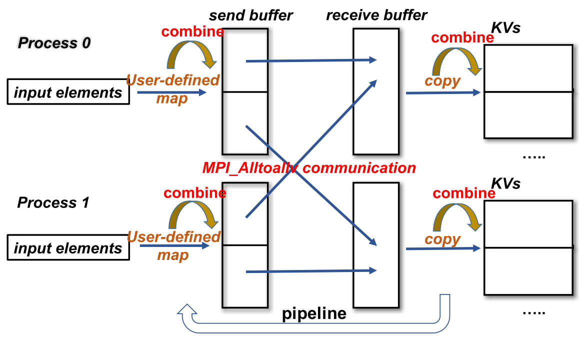


Fig. 4. Combiner workflow in Mimir with its `combiner` callbacks applied before and after each communication stage.

4.1 Combiner Workflow

From an implementation point of view, we extend MapReduce applications by allowing them to set a new combiner callback function that can be called within the map phase. The combiner callback takes two values as input and generates a single value as output. The pipeline combiner workflow is shown in Fig. 4. The figure points out how the `combiner` callback can be applied both before and after each shuffle communication stage of the map phase.

A combiner callback when applied before the communication stage reduces the communication size. On the other hand, a combiner callback when applied after the communication stage reduces the buffer size to store the KVs in the KVC. We structure the combiner workflow so that it is pipelined. Before any shuffle communication, KVs generated by the map callback are inserted into the corresponding partitions of the send buffer based on the partition function. When we encounter a KV with a key that is in the send buffer, the combiner callback is called. The combiner callback combines the two KVs (i.e., the new KV and the one already in the send buffer) into a single KV. The existing KV in the send buffer is then replaced with the combined version. For example, if in the word count benchmark execution the new KV is $\langle dog, 1\rangle$ and a KV with $\langle dog, 1\rangle$ is already in the send buffer, the replaced KV in the send buffer is $\langle dog, 2\rangle$.

After completion of a shuffle communication stage, the received KVs are inserted into the KVC. When a KV with a key that is already in the KVC occurs, the combiner callback is called. Similar to the combiner process before the shuffle communication stage, the combiner callback combines the two KVs into a single KV. The existing KV in the KVC then is replaced with the combined version. For example, if a process received a KV from a second process with $\langle dog, 2\rangle$ and already has the $\langle dog, 1\rangle$ KV, the execution of the callback replaces $\langle dog, 1\rangle$ with $\langle dog, 3\rangle$. The combiner callback is called multiple times—as many times as there are KVs with duplicate keys in the shuffle communication stages in the map phase.

To efficiently identify duplicate keys in KVs for each process, we use a hash bucket to track the position information of unique KVs in the send buffer and a hash bucket to track the position information of unique KVs in the KVC. In the case of the send buffer, we use the process's hash bucket to quickly check whether a key is already present in the send buffer. The hash bucket stores only the position information of the KVs; the actual KVs are still stored in the send buffer. The hash bucket of a process's KVC works in the same fashion. In our design of the combiner workflow, the two hash buckets require additional memory to keep track of the unique keys' positions. The spatial complexity of the two hash buckets is $O(u)$, where u is the number of unique keys.

4.2 Garbage Management

During the combiner optimization, the length of the combined KV may be different from that of the original KV in the send buffer or in the KVC. For example, the value field is a variable-length string. The combined KV can be stored in the same location where the original KV was, in cases where combining reduces the KV length. Otherwise, when the combined KV is larger than the original KV, the combined KV must be stored elsewhere. In either case, we will be left with some "holes" (i.e., garbage bytes) that do not contain useful data. We need to maintain a list of such unused spaces so that they can be reused for new KVs or can be reclaimed through garbage collection.

We introduce a hash bucket for each buffer to track these unused spaces. Each hole is represented in a $\langle startaddress, holelength\rangle$ format and is hashed by the start address. There are two hash buckets in each process for the buffers; one is used before and the other is used after the shuffle communication stage. In this way, only $O(1)$ time is needed to query whether a given address is the beginning of a hole. When Mimir needs to find a place to store a newly created or combined KV, it first tries to find an unused space that is large enough to fit the KV. If it is unable to find such space, it appends the KV to the end of the buffer. This approach allows us to reduce the total size of the garbage bytes.

Before starting a shuffle communication stage, Mimir may perform garbage collection to remove any existing holes and reclaim the space to reduce the data being transmitted during shuffle. The garbage collection phase scans the hash bucket and moves the KVs to cover the spaces that were holes. Since garbage collection is expensive, we perform it only when the accumulated size of the unused spaces is above a certain threshold. In the current implementation, we set this threshold to be the size of two pages. This also helps limit the memory usage of the hash bucket to a constant amount.

4.3 Memory Usage Analysis

To simplify the memory usage analysis, we again assume that the lengths of each key and each value are the same as we did in Section 3.3. We again use n to represent the total number of KVs and use u to represent the number of unique keys in those KVs. Note that u is always equal to or smaller than n. We have p processes to execute the MapReduce tasks. We further assume that the unique keys are partitioned evenly to all processes. The spatial complexity of the KVC after shuffle communication and the hash buckets to keep track of the unique keys is $O(u)$. The spatial complexity of the hash buckets to keep track of garbage bytes is $O(1)$. Thus, the overall spatial complexity of our pipeline implementation is $O(3 * u)$ (i.e., a KVC and two hash buckets for each process to keep track of the unique keys).

5 Data-Driven Optimizations

We present two data-driven optimizations: a dynamic repartition approach to mitigate the impact of the data skew problem

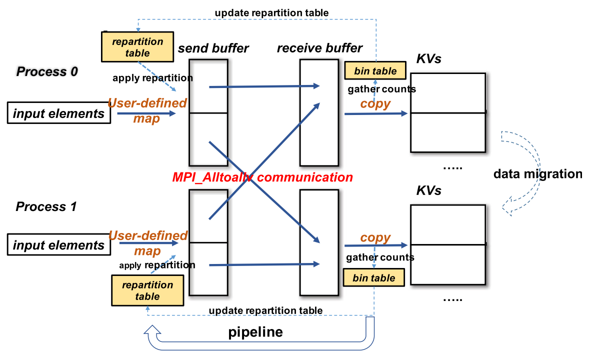


Fig. 5. Repartition workflow with the `repartition table` to allow repartitioned bins and `bin table` to record the KV counts of bins. KVs are grouped to bins in order to reduce the memory usage. KVs of repartitioned bins are migrated before the end of the map phase.

and a splitting strategy to deal with datasets in which a few keys occur significantly more frequently than do the other keys.

5.1 Dynamic Repartition Workflow

In order to ensure balanced memory usage across processes, each process should be assigned a similar number of KVs during the shuffle communication stage. The difficulty of partitioning KVs evenly is that the distribution of KVs is unknown before the MapReduce processing starts. Traditionally, MapReduce frameworks use the hash value of the key to partition KVs. However, this hash partition cannot balance the KVs for highly skewed datasets. To overcome this limitation, we first partition the KVs based on the hash values of the keys and then dynamically adjust the partition by repartitioning some keys as the MapReduce framework gathers more information about the distribution of its KVs. Note that our repartition method is transparent to the MapReduce application's execution.

Our dynamic repartitioning is implemented in the map phase. The workflow is shown in Fig. 5. To record repartitioned keys and keep track of KV counts, we introduce two new data structures: the `repartition table` and the `bin table`. The `repartition table` consists of a list of repartitioned bins and the rank of the process owning that bin. The `bin table` counts all KVs whose keys are grouped into a bin. We use the bin as the minimum repartition unit, rather than the individual keys contained in the bin, in order to further reduce the amount of memory used. In other words, we might consume enormous amounts of memory if we keep track of information for each unique key. The bin index for one key is $hash(key)\%(b*p)$, in which b is a configurable parameter called the bin count and p is the number of processes. Both the `repartition table` and the `bin table` are implemented as hash tables. Thus, the time complexity to get the value from either table is $O(1)$.

The map phase starts with the computation stage and generates KVs by the map callback. Once a KV is generated, the way to compute the target process of the KV is different from the basic workflow in other implementations of MapReduce over MPI such as the original Mimir [15] and MapReduce-MPI[27]. Specifically, the `repartition table` is searched first. If the bin that contains this KV is found, then the target process rank is obtained from this table. Otherwise, the target process is computed by $(binindex)\%(p)$. After getting the target process, the KV is inserted into the corresponding partition of the send buffer. If any one partition is full, the shuffle communication stage starts, and the KVs are exchanged. When a process receives a set of KVs from other processes, it gathers the counts of all KVs into its bin table before saving the information into the KVC. The load imbalance is assessed at regular shuffle communication intervals called the check frequency. If no load imbalance is found, the MapReduce workflow resumes the computation stage. Otherwise, the repartition algorithm is executed to update the `repartition table.` We perform a *lazy* intermediate data migration strategy; that is, KVs that belong to repartitioned bins are migrated to new target processes after all processes finish the map computation.

Our load-balancing check, repartition algorithm, and intermediate data migration methods are discussed in the next three subsections.

5.1.1 Load-Balancing Check

Each process uses a variable to keep track of the number of received KVs. The load-balancing check is performed by

$$(maxcount)/(mincount) > balance factor,$$

in which *max count* and *min count* are the maximum and minimum number of KVs received across processes, respectively. The *balance factor* is a configurable parameter to determine whether to perform the repartition: the program performs the repartition if the (max count)/(min count) is larger than the balance factor. The load-balancing check is implemented by using two `MPI_Allreduce` calls to get the max and min counts separately. Each process executes the check algorithm independently.

5.1.2 Repartition Algorithm

Once we identify the load imbalance of the KVs, we execute the repartition algorithm to update the process's `repartition table`. When performing a repartition, we may have only partial count information of the entire dataset's KVs because the MapReduce job may be in progress. To deal with such partial knowledge, we base our repartition algorithm on the sampling principle in [10]: the frequency distribution of each key in the partial dataset is used to predict the frequency distribution of the full dataset. For example, assume that completing the processing of a given input dataset requires ten communication stages and that we identify the load imbalance problem at the end of the second shuffle communication stage. Under these circumstances, we execute the repartition algorithm with information on only approximately 20 percent of the total KVs generated. In our example, additional load imbalance may occur, for example, after the sixth shuffle communication stage. In that case, we perform another repartition in the sixth shuffle stage to adjust the previous repartition, but this time with information from about 60 percent of the total KVs. The key idea here is that as the execution progresses, our repartitioning becomes increasingly more accurate. But at the same time, we may need to migrate more KVs in those later stages. By performing a repartition each time a load imbalance problem is detected, we aim to reduce the intermediate data migration overhead in the later stages.

For efficient memory usage, we use bins as the repartition unit, and a key that belongs to a given bin is always assigned to the same process. Our repartition algorithm uses the `bin table` to predict the distribution of KVs belonging to different bins. The repartition problem is similar to the bin-packing problem. Let us assume that we have $b * p$ bins and that we know the estimated frequency of each bin. Our goal is to assign the $b * p$ bins to p partitions (where p is also the number of processes) in as balanced a way as possible. This problem is NP-complete. We address it by using a heuristic algorithm with two steps. In the first step, the total repartition percentages of the different processes are computed. For example, if we have two processes, Process 0 and Process 1, and if Process 0 has received 75 percent of the KVs and Process 1 has received 25 percent, then we know that Process 0 needs to forward 25 percent of its KVs to Process 1. In the second step, each process identifies those bins that are to be exchanged in the repartition process. In our example, if Process 0 has three bins with frequencies of 50, 15, and 10 percent, then Process 0 repartitions the bins with frequencies of 15 and 10 percent to Process 1.

5.1.3 *Intermediate Data Migration*

Before the end of the map phase, we need to migrate the KVs of the repartitioned bins that were identified by the repartition algorithm to the new target processes. If Key s is partitioned and assigned to Process i before a repartition and assigned to Process j after a repartition, then no further KVs with Key s will be sent to Process i. Thus, we need to migrate all the KVs to Process j before the end of the map phase to avoid having some KVs with the same key end up in different processes. Because a bin is our atomic unit of KVs, for each repartitioned bin we need to migrate the corresponding KVs to the new target process. We use a lazy intermediate data migration strategy.

To integrate our migration methods in Mimir, we extend the KVC to support a `remove` interface. The KVs in the KVC are scanned one by one. If a KV that belongs to a repartitioned bin is found, then the `remove` interface is invoked to remove the specific KV from the KVC. Once a KV is removed, the bytes of the KV in the KVC are marked as a "hole" (i.e., garbage bytes) which can be reclaimed through the garbage collection process that we explained in Section 4.2.

To send the KVs of the repartitioned bins to new target processes, we reuse the communication buffers (i.e., send buffer and receive buffer) in the shuffle communication stages. Thus, we do not introduce any extra memory usage for migrating KVs.

5.1.4 *Memory Usage Analysis*

The extra memory usage for our repartition design includes the memory for the `repartition table` and `bin table`. The length of the `repartition table` is $x * b * p$, where x is the percentage of repartitioned bins, b is a configurable parameter, and p is the number of processes. The maximum length of the `bin table` in any process is $b * p$. Since x, b, and p are much smaller than n for large-scale datasets, the spatial complexity of the dynamic repartition is still $O(n)$. The extra memory usage is significantly smaller than the memory usage of intermediate data, indicating that our approach does not impact the overall memory usage of MapReduce applications.

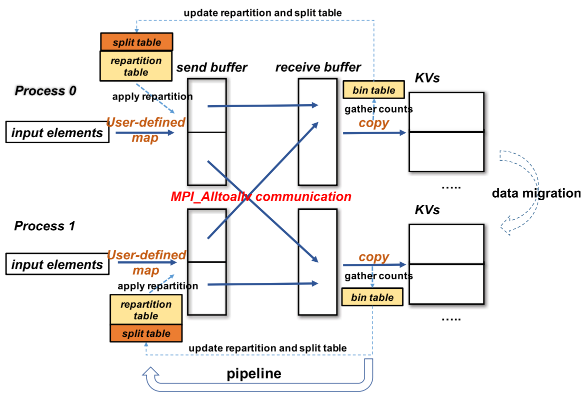


Fig. 6. Split workflow with `bin table` to record the KV counts of bins, `repartition table` to record repartitioned bins, and `split table` to record superkeys. KVs of superkeys are distributed to all processes.

5.2 Superkeys and Splitting Strategy

The previous repartition design can balance the memory usage well for most situations. In some situations, however, some keys appear significantly more frequently than do other keys. We call these keys *superkeys*. The standard MapReduce workflow requires that the KVs with the same key be sent to the same process. Thus, balancing the number of KVs is impossible when one or two superkeys occur in a largely diverse dataset. For example, if the KVs with one key make up 5 percent of all KVs and we use 25 processes to handle this dataset, then this dataset cannot be balanced because, in order to be fully balanced, each process cannot have more than 4 percent of the KVs. Moreover, as we use more and more processes, a more serious load imbalance problem arises when superkeys are present. For our example, if we use 1,536 processes, then for perfect load balancing the average percentage of keys in each process is 0.065 percent. If we assign one superkey with a frequency of 5 percent to one process and balance all the other KVs evenly among the remaining processes, then each of the other processes is assigned about 0.061 percent of the KVs, which is nearly 82 times smaller than the number of KVs on the process to which the superkey was assigned.

To solve this problem, we devise a split method that enables the Mimir workflow to deal with superkeys. Our split method provides three features: (1) it ensures that the number of split keys is small; (2) it provides the capability to get the split key list; and (3) it balances the KVs with the split key among all processes. These features are important for split-tolerant algorithms to handle the KVs with split keys. Key splitting is an extension to the standard MapReduce workflow; thus the application needs to enable it using a runtime option. By making key splitting optional, we allow applications that have potential superkeys to benefit from it while avoiding such overhead when key splitting is not needed.

First, we extend the repartition workflow described in Section 5.1 to support splitting. Our split workflow is shown in Fig. 6. We add a `split table` to record split keys. Note that we record the hash value of the split key instead of the key itself so that we can further reduce the overhead involved in searching for the key when the length of the key is long. The `split table` is used when computing the target process. If the hash value of the key is found in the `split table`, then the target process is randomly chosen (i.e., the KVs with split key are distributed evenly among all processes). Otherwise, the target process is computed in the same way as the repartition workflow described in Section 5.1.

Second, we design the algorithm to get the split keys after executing the repartition algorithm. The `bin table` is scanned to find suspect bins that may contain superkeys, that is, bins that contain $1/p$ KVs of the total number of KVs. We then scan the KVs one by one, recording the KV count of each key that belongs to the suspect bins. Once we have all KV counts belonging to suspect bins, the keys that contain at least a certain proportion of all KVs are added to the `split table`. The proportion is set to $0.8/p$ by default. At the same time, the split keys are recorded in order to allow MapReduce applications to access them after the map phase.

If we assume that there is no hash collision, the maximum number of split keys is $1.25 * p$. Thus, the worst-case spatial complexity of the `split table` is $O(p)$. Our results in Section 7 demonstrate the effectiveness of our splitting algorithm with MapReduce applications such as *join* that traditionally cannot handle datasets with superkeys efficiently.

6 Integration Into Unified Framework

We integrate our dynamic repartition workflow in Section 5.1 into the combiner workflow that supports merging KVs with the same key in Section 4 to create a unified and powerful MapReduce over MPI framework. The novelty of our work lies in the merging of combiner optimizations with methods for balancing memory usage skew. Contrary to other MapReduce implementations [11], [31], [37], in our implementation we balance the number of unique keys rather than the total number of KVs to balance the memory usage. To this end, we replace the number of KVs in the repartition workflow described in Section 5.1 with the number of unique keys. Earlier, in Section 4.3, we showed that the spatial complexity of the combiner workflow is $O(u)$. To check for load imbalance, we count the number of unique keys assigned to each process. The `bin table` is used to keep track of the unique keys belonging to each bin. We modify the repartition algorithm described in Section 5.1.2 to use the updated `bin table` to partition unique keys evenly. The basic design of the migration algorithm remains the same as in Section 5.1.3, except that combined KVs are migrated.

7 Evaluation

In this section, we evaluate the results of Mimir's in-memory workflow, the combiner workflow, the dynamic repartition workflow, and the superkey and splitting approach with respect to memory usage and performance (i.e., execution time).

7.1 Platforms, Benchmarks, and Datasets

Our tests are performed on the Tianhe-2 supercomputer. Tianhe-2 is a high-performance supercomputer located at the National Supercomputer Center in Guangzhou, China. Each compute node has two Intel Xeon E2-2692v2 CPUs (i.e., 12 cores each, 24 cores total) running at 2.2 GHz. Each node has 64 GB of memory. The nodes are connected with Tianhe express-2 [22], and the parallel file system is H2FS [33]. We use MPICH 3.1.3 with a customized GLEX channel on Tianhe-2 [32]. We perform tests on single nodes and across nodes of Tianhe-2.

We have presented two classes of optimizations: combiner optimizations and repartitioning optimizations. We use three benchmarks that are diverse in terms of datasets and their features: *word count* (WC), *octree clustering* (OC), and *join* (Join). WC is an example application that can benefit from the first optimization. OC clustering and Join are example applications that can benefit from the second optimization.

WC is a single-pass MapReduce application. It counts the number of occurrences of each unique word in given input files. OC is an iterative MapReduce application with multiple MapReduce stages. As the application name suggests, OC is essentially an interactive clustering algorithm (a chain of MapReduce jobs) for points in a n-dimensional space. We use the MapReduce algorithm described by Estrada et al. [14] for classifying points representing ligand metadata from protein-ligand docking simulations with its 3-D data points. Join is a single-pass MapReduce application that merges two datasets into one dataset. Among various join applications, we choose the log-processing application described in [7], which joins a larger dataset (a log dataset) and a smaller dataset (a reference dataset). For the MapReduce implementation, we use the algorithm called *repartition join* described in [7]. In the map phase, the $\langle key, value\rangle$ pairs are generated from the two datasets. To identify which dataset a KV pair is from, we add a tag to the value field to mark the original dataset. All KV pairs with the same key are then grouped and passed to the same reduce function. The reduce function buffers the KV pairs from the smaller dataset and then performs a cross-product between records in the two datasets. During the evaluation, the smaller dataset is always eight times smaller than the larger dataset; we report only the larger dataset in the figures.

We generate both balanced and imbalanced datasets. We work with synthetic datasets whose elements (words for WC and Join, and n-D points for OC) are generated following the Zipf distribution [4]. We choose the Zipf distribution because it is common in real-world data, such as word counts in the Wikipedia dataset or city sizes. The Zipf distribution has two parameters: n and α; n is the number of unique items in the dataset, and α represents the degree of skew. If α is 0, the distribution of items (i.e., words and points) is balanced. As α increases, the degree of skew and the imbalance increase. When dealing with imbalanced datasets, we consider two sources: value imbalance (i.e., the imbalance is reflected in the value distribution) and key-mapping imbalance (i.e., the imbalance is reflected in the unique key distribution across processes). An example of value imbalance in the WC benchmark is as follows. Suppose we have multiple KV pairs $< k1, v1 >$, $< k1, v2 >$, $< k2, v3 >$ and we partition them between two processes. If the keys are partitioned evenly (i.e., $k1$ to one process and $k2$ to the another process), then the dataset is value imbalanced across processes (i.e., $< k1, v1 >$ and $< k1, v2 >$ to one process and $< k2, k3 >$ to another process). On the other hand, an example of key-mapping imbalance for the same benchmark is as follows. Suppose we have multiple KV pairs $< k1, v1 >$, $< k1, v2 >$, $< k2, v3 >$,

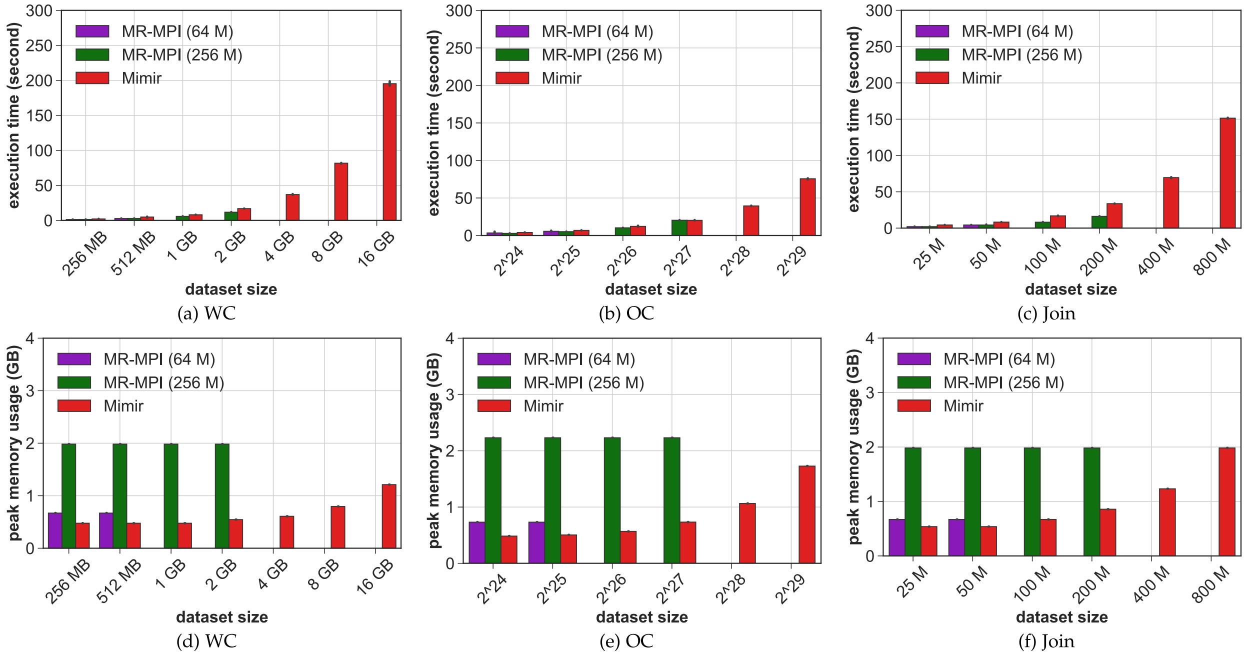


Fig. 7. Single-node results with balanced datasets ($\alpha = 0.0$).

and $< k2, v4 >$ and we partition the dataset between two processes. If the keys are partitioned unevenly (e.g., $k1$ and $k2$ all to one process), then the dataset is key-mapping imbalanced across processes. This situation is possible when the hash function used to assign KVs to processes is not able to balance the keys.

Our metrics of success are execution time and peak memory usage. For imbalanced datasets, we also track the memory balance ratio. Execution time is the time from reading the input data to outputting the final results of a benchmark. The input data and output data are stored in the parallel file system of Tianhe-2. Peak memory usage is the maximum memory usage at any point in time during the benchmark execution. Memory balance ratio is the maximum peak memory usage across all processes divided by the minimum peak memory usage across all processes for the execution. Tests are performed three times, and standard deviations are reported in the figures.

7.2 Balance Versus Imbalance In-Memory Runs

We consider the implementation of Mimir presented in Section 3 with its in-memory optimizations as our "baseline" implementation and compare it with MR-MPI in terms of execution time and peak memory usage on a single node; we also measure the weak scalability on multiple nodes. The page size of MR-MPI is set to 64 MB, the default, and to 256 MB, the maximum size possible for MR-MPI pages on Tianhe-2, so that MR-MPI can use all of the memory on the platform.

Fig. 7 shows the execution times and the memory usage for all three benchmarks running on a single node of Tianhe-2 with balanced datasets (i.e., with skew degree equal to 0.0). As long as the datasets fit in memory, the execution times of the two frameworks (i.e., Mimir and MR-MPI) are comparable, with MR-MPI performing slightly better than Mimir for some of the benchmarks. This result is expected because the focus of Mimir is on processing more data in memory than what MR-MPI can. Once a dataset can no longer be hosted in memory, the performance for the impacted benchmark degrades substantially (execution times become several orders of magnitude higher). In our tests, these out-of-memory execution scenarios are observed for MR-MPI; in our figures, the degraded MR-MPI measurements are not reported.

Across the three benchmarks and their different datasets, Mimir always uses less memory than MR-MPI does: at least 20 percent less memory compared with MR-MPI with page size 64 MB and at least 56 percent less memory compared with MR-MPI with page size 256 MB. As pointed out in the preceding paragraph, when datasets increase in size, MR-MPI runs out of memory. Our results for WC indicate that for datasets larger than 512 MB, MR-MPI with a page size 64 MB runs out of memory, and for datasets larger than 2 GB, MR-MPI with page size 512 MB runs out of memory. Mimir, on the other hand, supports in-memory analysis for up to 16 GB datasets—8-fold larger than the best case of MR-MPI. This improvement is due solely to Mimir's workflow, which uses memory more efficiently. For OC and Join, Mimir still uses less memory and supports in-memory computation for larger datasets than does MR-MPI. Specifically, Mimir allows execution of datasets 4-fold larger for both OC and Join compared with MR-MPI with page size 526 MB and 16-fold larger for both OC and Join compared with MR-MPI with page size 64 MB.

We perform similar comparisons for the three benchmarks with skewed datasets, that is, with both value and key-mapping imbalance. Similar patterns as for the balanced datasets are observed for WC, OC, and Join when using the imbalanced datasets: (1) Mimir and MR-MPI have similar execution times; (2) MR-MPI runs out of memory even for smaller datasets than for the balanced datasets; and (3) Mimir supports the same large datasets as for the balanced scenarios. Fig. 8 shows the results for the value-imbalanced scenarios with degree of skew equal to 1.0. For WC and Join, MR-MPI with page size 64 MB cannot execute in memory even for small datasets; for OC, MR-MPI with page size 64 MB cannot run for datasets larger than 2^{25}. The reason is that the skewness in datasets causes some

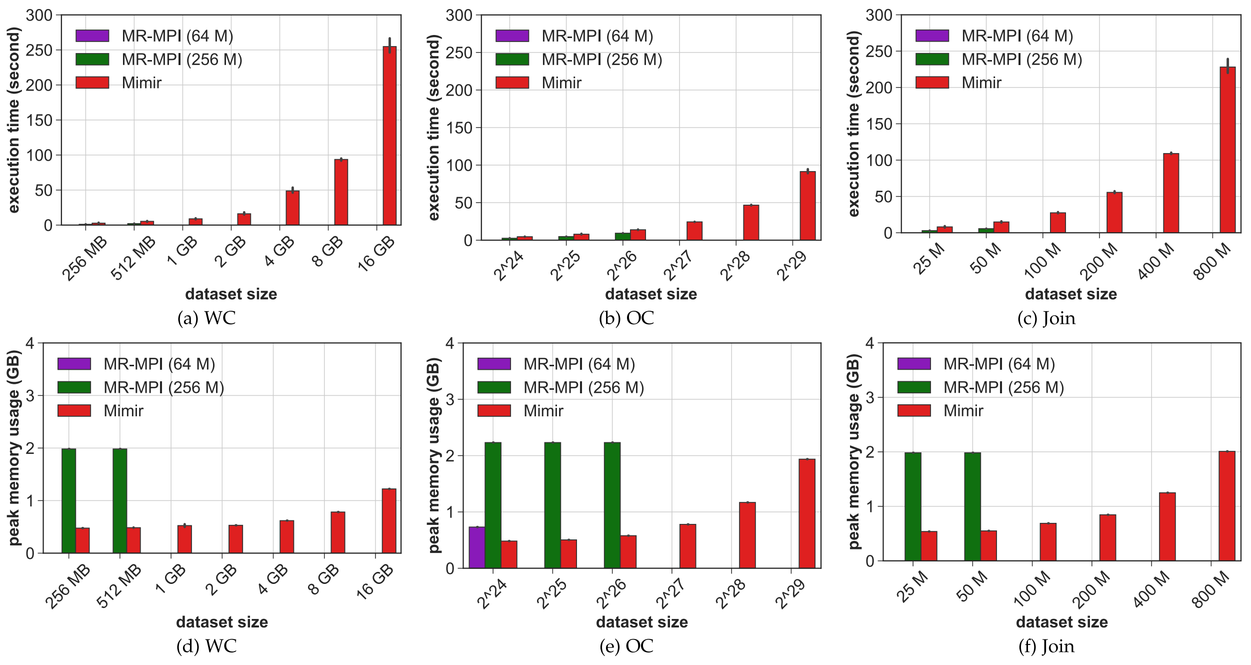


Fig. 8. Single-node results with imbalanced datasets (value imbalance $\alpha = 1.0$).

processes not to have enough memory for execution. For WC, OC, and Join, Mimir uses less memory for small datasets and allows execution of larger datasets—32-fold larger for WC, 8-fold for OC, and 16-fold larger for Join compared with MR-MPI with page size 256 MB. For space constraints we omit the figures for the key-mapping-imbalanced scenarios in which we further observe the shrinking of the datasets (in terms of their size) supported by MR-MPI with page size 256 MB. For key-mapping-imbalanced data, Mimir can process 64-fold larger datasets than MR_MPI can for WC, and 32-fold larger for OC and Join.

We compare the weak scalability of Mimir versus MR-MPI for the three benchmarks on 1, 2, 4, 8, 16, 32, 64, and 128 nodes (i.e., 24, 48, 96, 192, 384, 768, 1536, and 3072 cores, respectively) when using both balanced and imbalanced datasets.

For the balanced datasets, we run two sets of tests with a small and a large dataset, respectively. In the first set of tests we keep the input data size per node to the maximum size that MR-MPI with page size 256 MB configurations can run on a single node, as reported in Fig. 7 (i.e., 2 GB for the WC, 2^{27} for OC, and 200 M for Join). In the second set of tests, we keep the input data size per node to the maximum size Mimir can support (i.e., 16 GB for WC, 2^{29} for OC, and 800 M for Join). For the imbalanced datasets we still consider both a small dataset and a large dataset. The small dataset further shrinks to meet the maximum size that MR-MPI with page size 256 MB configurations can support on a single node with the value-imbalanced datasets (i.e., 512 MB for the WC, 2^{26} for OC, and 50 M for Join) and with key-mapping-imbalanced datasets (i.e., 256 MB for the WC, 2^{25} for OC, and 25 M for Join), as defined in the preceding section. For the large dataset, we keep the input data size per node to the maximum size Mimir can support for both value-imbalanced datasets (i.e., 16 GB for WC, 2^{29} for OC, and 800 M for Join) and key-mapping-imbalanced datasets (i.e., 16 GB for WC, 2^{28} for OC, and 400 M for Join).

We observe that Mimir can scale up to the 3,072 cores of Tianhe-2 independently from the benchmarks when using small datasets (i.e., for balanced—not shown in the paper for space constraints, for value imbalance as shown in Figs. 9a, 9b, 9c, 9d, 9e, and 9f, and for key-mapping imbalance as shown in Figs. 10a, 10b, 10c, 10d, 10e, and 10f).[1] While MR-MPI with page size 256 MB can also scale up to 3,072 cores for the same balanced datasets, its scalability is limited to up to 96 cores for imbalanced datasets as shown in Figs. 9a, 9b, 9c, 9d, 9e, 9f, and 10a, 10b, 10c, 10d, 10e, 10f. This loss in scalability is due to some processes having more intermediate data and thus exceeding the 256 MB page size and spilling to the I/O subsystem.

When using the large datasets, MR-MPI spills to the I/O subsystem on a single node (as shown in Figs. 7 and 8) and therefore does not scale. Mimir scales up to 768 cores for WC, 384 cores for OC, and 192 cores for Join when using value-imbalanced data, as shown in Figs. 9g, 9h, 9i, 9j, 9k, and 9l; it scales up to 48 cores for WC, 96 cores for OC, and 96 cores for Join when using key-mapping-imbalanced data, as shown in Figs. 10g, 10h, 10i, 10j, 10k, and 10l. Clearly, Mimir's scalability outperforms MR-MPI for both small and large datasets. The optimizations in Sections 4, 5.1, and 5.2 enable the further increase of Mimir's scalability presented in the next section.

7.3 Impact of Optimizations

We measure the impact of the workflow optimizations that we present in Sections 4, 5.1, and 5.2 on scalability and performance of WC, OC, and Join, when using skew datasets (i.e., both value-imbalanced and key-mapping-imbalanced datasets). Fig. 11 shows the weak scalability for the value-imbalanced datasets (i.e., the execution times, the peak memory performance, and the memory balance ratio) for our

1. We note that 3,072 is the maximum number of cores available to us for our tests.

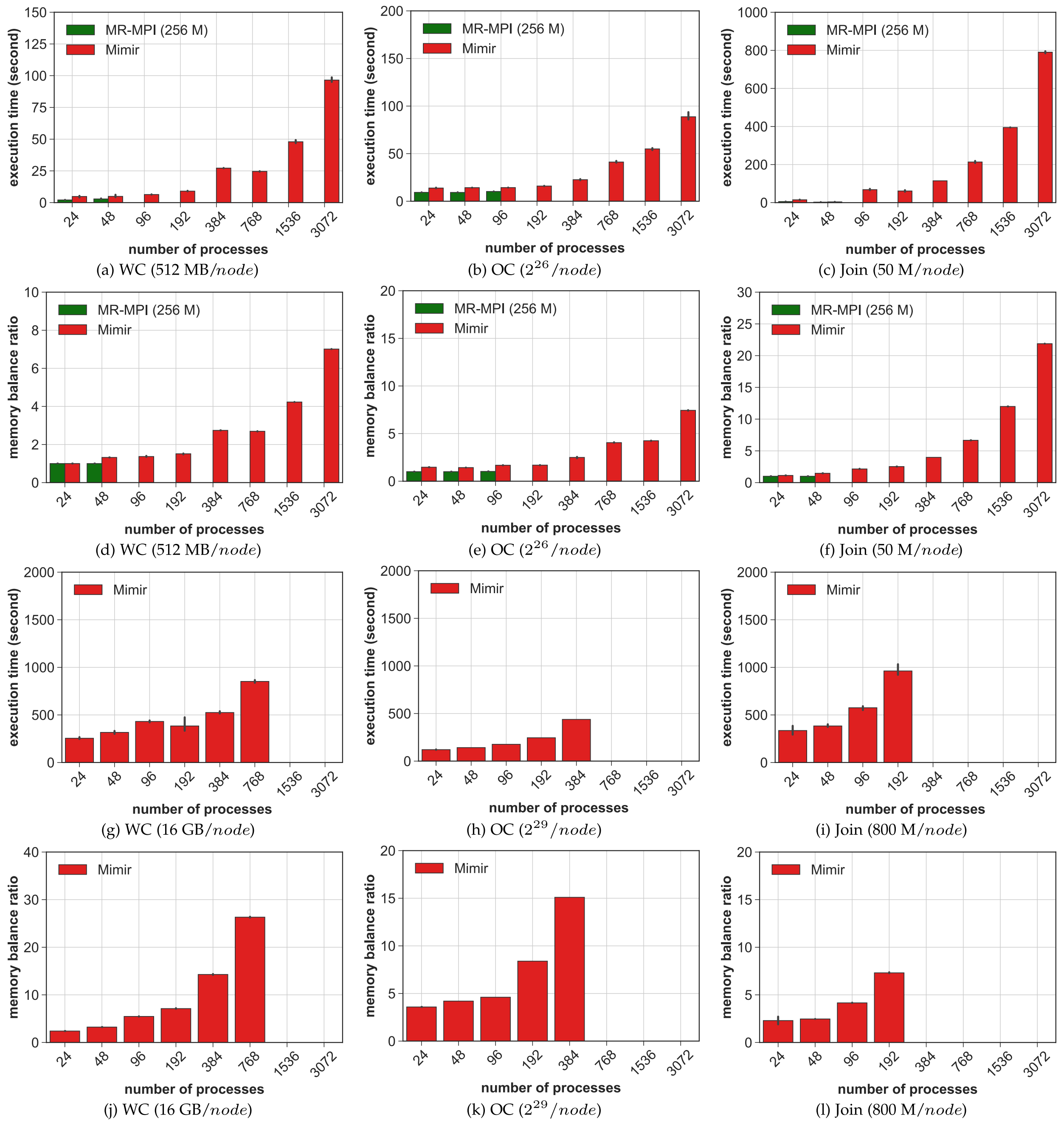


Fig. 9. Weak-scalability results with imbalanced datasets (value imbalance $\alpha = 1.0$).

baseline Mimir described in Section 3; Mimir with combiner optimizations (i.e., Mimir+cb) presented in Section 4; and Mimir with dynamic reparation (i.e., Mimir+cb+rp) and with superkey and splitting strategies (Mimir+cb+rp+sp) presented in Sections 5.1 and 5.2). Fig. 12 shows the weak scalability for the same optimizations but for key-mapping-imbalanced datasets.

In the two figures, we observe how different optimizations are beneficial for different benchmarks and datasets. In Fig. 11, combiner optimizations can balance the memory usage, decrease the memory requirement, and consequently reduce the execution times for WC and OC when using value-imbalanced datasets. Moreover, the same optimizations substantially increase the scalability of the two benchmarks by reducing the amount of data that are moved across processes in the shuffling phase. Further optimizations for the two benchmarks, namely, dynamic repartition and superkey splitting strategies, do not have any further impact on the two benchmarks.

When considering key-mapping-imbalanced datasets, one must notice that combiner optimizations depend on the hash function to distribute the unique keys evenly across different processes. Hash functions may not distribute unique keys evenly in cases such as when keys are not uniform in the hash space (i.e., for the key-mapping imbalance). This concatenation of facts explains why the same two benchmark do not present the same performance and scalability improvements for the key-mapping-imbalanced data presented in Fig. 12. In other words, for key-mapping-imbalanced datasets, the default hash function cannot

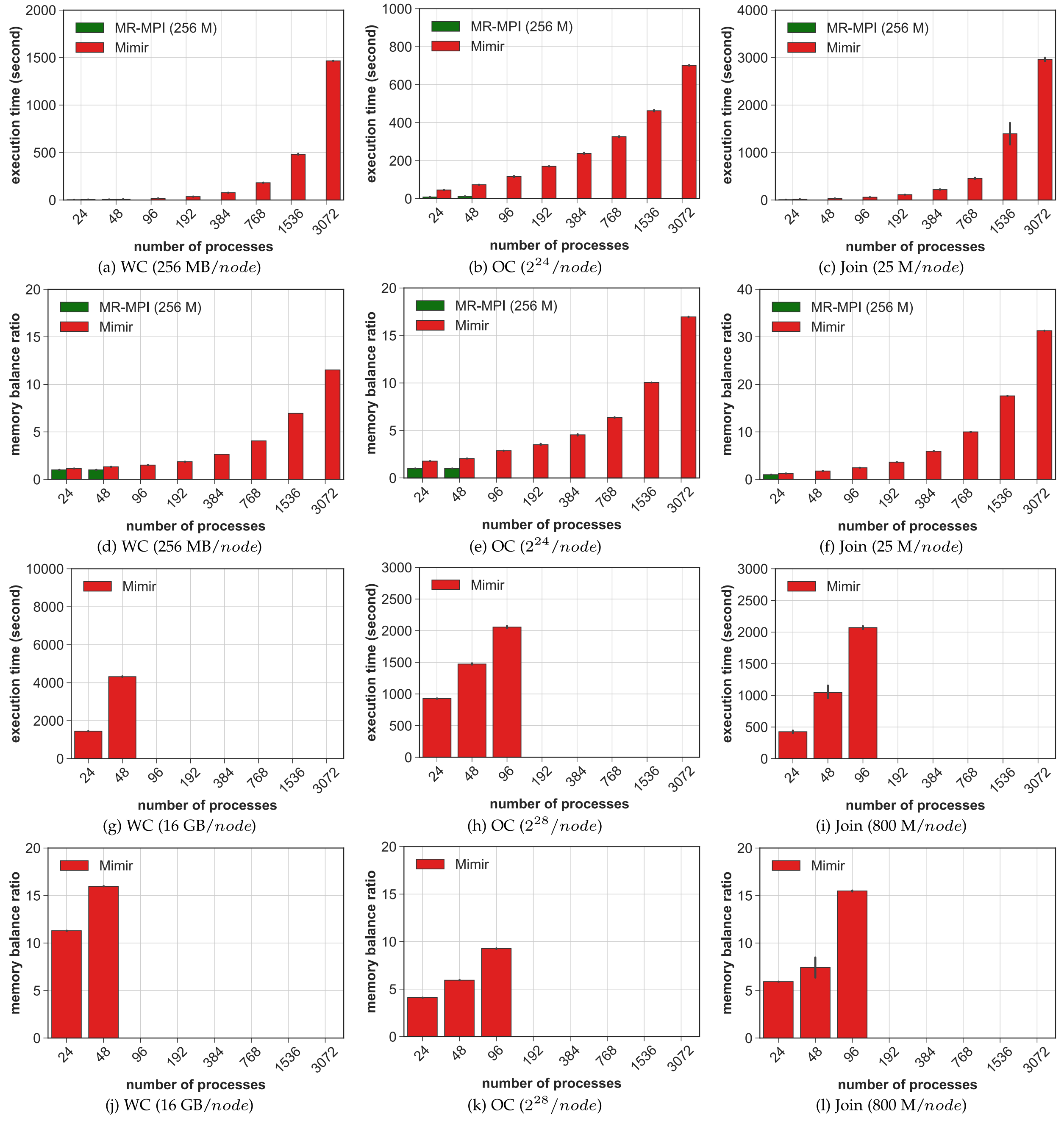


Fig. 10. Weak-scalability results with imbalanced datasets (key-mapping imbalance $\alpha = 1.0$).

distribute the unique keys evenly, whereas the hash function could balance the unique keys for the value-imbalanced datasets. Further optimizations are needed for the key-mapping-imbalanced datasets such as the dynamic repartitions that actively redistribute the data across processes. Our dynamic repartition balances the memory usage well and improves the performance of the key-mapping-imbalanced dataset up to 13 times, as shown in Fig. 12. Thus, the results prove the effectiveness of integrating our repartition with the combiner in order to solve these abnormal situations associated with the hash function's inability to balance unique keys. Further optimizations for the two benchmark with superkey strategies do not have any further impact on the benchmarks' performance.

Join exhibits different patterns from those of WC and OC. Join datasets have a log-processing distribution. In Fig. 11, a severe imbalance due to the long tail of the datasets results in poor scalability for Join when using a value-imbalanced dataset. The problem can be resolved for these value-imbalanced scenarios only with the splitting strategies defined in Section 5.2. Our splitting strategies rebalance the KVs, resulting in memory balance and better performance. The same behavior is not observed for Join in Fig. 12 when using key-mapping-imbalanced datasets for which the benchmark scalability is improved already with the dynamic reparation optimizations. One can clearly see that while the scalability improves, the execution times seem to suffer from overhead associated with the dynamic reparation of data chunks.

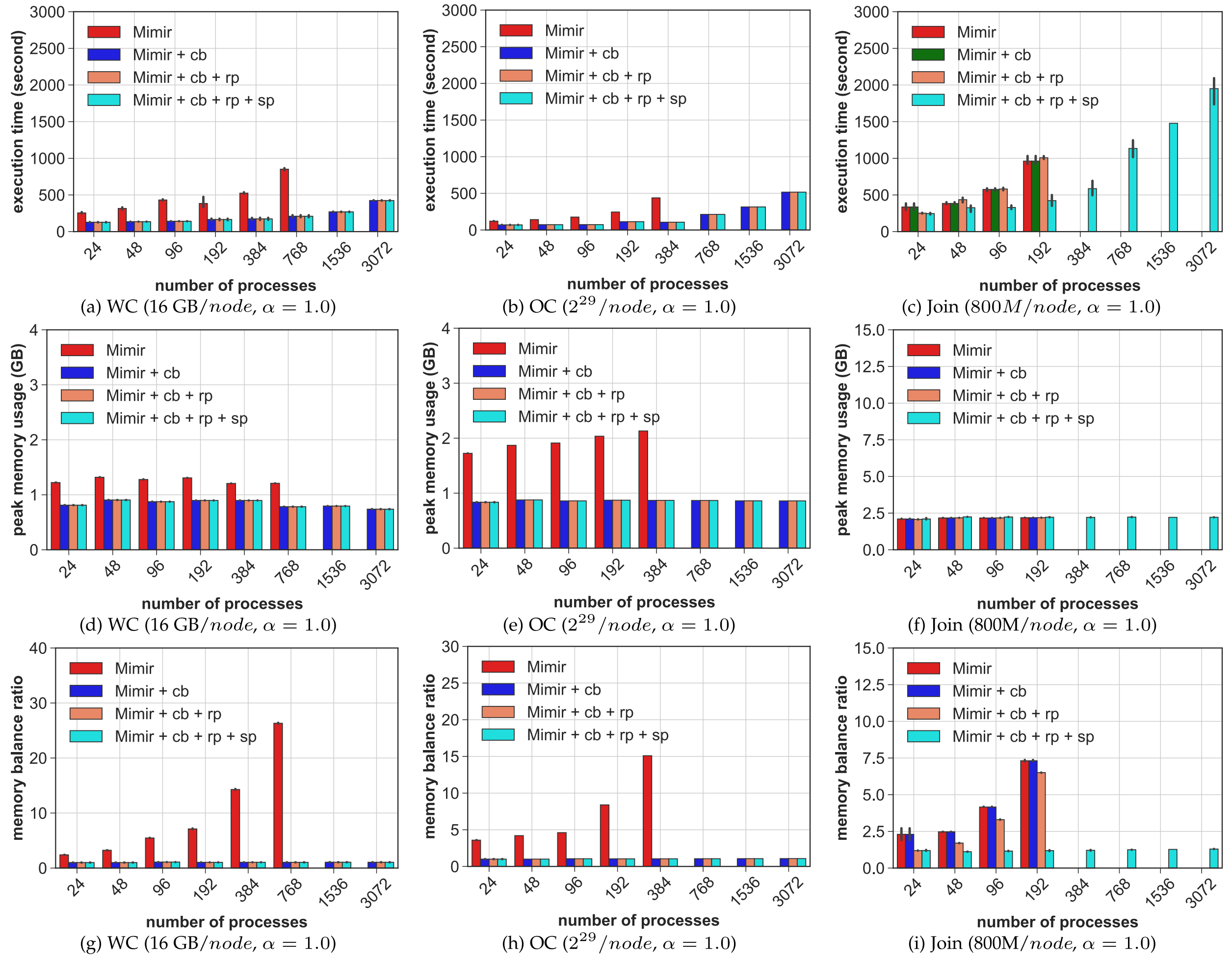


Fig. 11. Weak-scalability results for the different optimizations and with imbalanced datasets (i.e., value imbalance).

Table 1 summarizes the maximum number of cores for which each benchmark scales for both the value-imbalanced and key-mapping-imbalanced datasets when the different optimizations are applied. In bold we also outline the best performance observed (i.e., execution times) for each type of optimizations. The table supports our claim that different benchmarks and datasets need different types of optimizations.

8 Related Work

Most studies of data skew migration in the MapReduce environment [10], [16], [17], [19], [20] focus on MapReduce frameworks designed for cloud and commodity cluster systems. Different from them, our work is to mitigate the skew in MapReduce for supercomputing systems. For example, in [28], [30], the authors optimize Hadoop and Spark on commodity clusters using RDMA techniques. While these techniques take advantage of modern network capabilities to improve MapReduce, they are fundamentally limited to frameworks such as Hadoop and Spark that assume a commodity cluster model, (that is, a model in which each node is equipped with on-node storage that helps with storing temporary files, as needed). While this does not entirely avoid the skew problem and thus requires additional improvements, the general environment is different enough that these solutions cannot directly be applied to supercomputing systems that have no on-node storage.

In the initial MapReduce implementation of Google [11], Dean and Ghemawat used speculative execution to mitigate the time skew. The speculative execution method has been improved [6], [9], [38]. However, this method cannot handle the situation in which one reduce task is assigned too much data because of data skew. Instead, our design handles the load imbalance problem caused by the data skew.

Other work is based on the static partition method and aims to find a better partition for the intermediate data. SkewReduce [19] proposes a cost-based partitioning optimization that allows the user to provide a cost function instead of simply partitioning the data evenly. This is useful for some applications in which the execution time is not linearly decided by the data size. However, SkewReduce depends on a sampling program to get the approximate frequencies of keys. Thus, it introduces extra overhead to run the sampling program. Shadi et al. [17] propose an algorithm to consider the locality and fairness to reduce the data movement over network. However, they need to delay the shuffle communication until all map tasks are done, in order to gather the frequencies of all keys. TopCluster [16] proposes a method to estimate the frequencies of top-k keys. This method is useful because getting accurate frequencies of all keys for large-scale datasets is not scalable. A similar idea is used by Yanfang et al. [21]. LIBRA [10] is also based on the sampling method, which the authors improve by

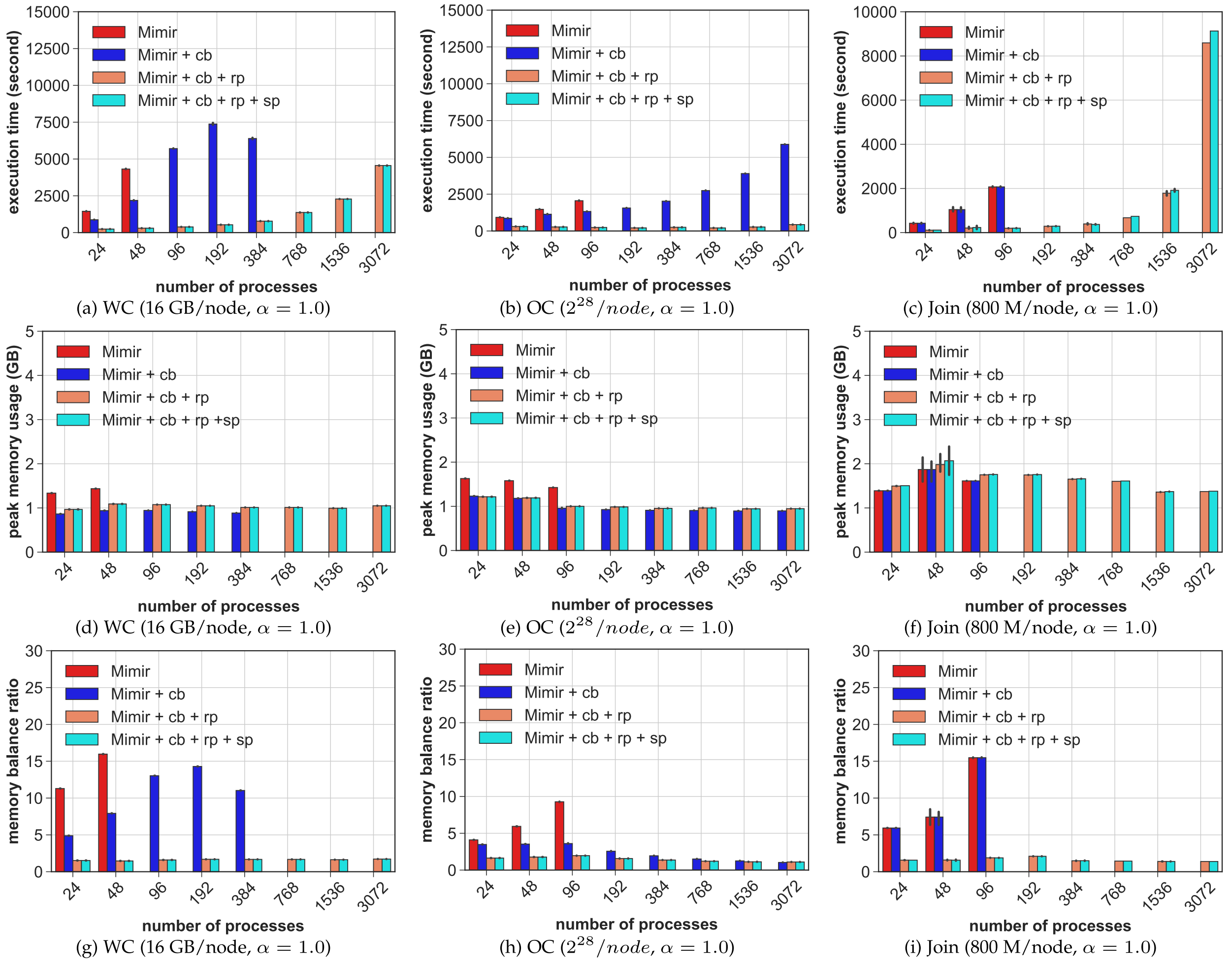


Fig. 12. Weak-scalability results for the different optimizations and with imbalanced datasets (i.e., key-mapping imbalance).

integrating sampling into a small percentage of the map tasks. Thus, they avoid the need to run sampling programs and do not need to delay all shuffle communications. Different from these static partition methods, our repartition design can dynamically adjust the partition. More important, we do not delay any shuffle communications or run any sampling programs.

Other methods use dynamic partitioning. Yanfang et al. [21] propose an online partition method based on a greedy strategy that assigns each key to the task with the smallest load. This method depends on the master-worker design, however, and is not suitable for the decentralized MapReduce design. SkewTune [20] is another system that adjusts the partition dynamically. The basic idea is to repartition unprocessed data of straggler tasks. Our dynamic repartition method has some fundamental differences compared with that idea. First, our repartition method is based on the sampling principle: that is, our method tries to use the frequencies of a partial dataset to estimate the frequencies of the entire dataset. Thus, we can eliminate the load imbalance problem as early as possible. Second, SkewTune cannot repartition any assigned keys for the reduce tasks, whereas our method can reassign keys by migrating intermediate data.

TABLE 1
Maximum Scalability of Mimir in Terms of Number of Processes for the in-Memory Workflow, the Combiner Workflow (cb), the Dynamic Repartition Workflow (rp), and Splitting Approach (sp)

	Value			Key Mapping		
	WC	OC	Join	WC	OC	Join
Mimir	798	384	192	48	96	96
Mimir+cb	**3072**	**3072**	192	384	3072	96
Mimir+cb+rp	3072	3072	192	**3072**	**3072**	**3072**
Mimir+cb+rp+sp	3072	3072	**3072**	3072	3072	3072

Numbers in bold represent the configuration with best performance after which further optimizations do not increase the performance.

Some research also has been carried out to handle superkeys. For example, LIBRA [10] introduces a split design. However, none of the previous works provides the split key list to applications. Thus, they do not allow the applications to handle the split key and nonsplit key separately, as in our design.

Data skew has also been studied in the parallel database area for Join [34], [35], group [3], aggregate [26], and so on. Recent work [12] about adaptive query processing focuses on relational operators [12]. Our work is more general, however, by supporting MapReduce applications.

Data skew or load-balancing problems have also been studied in scientific simulation communities. As with parallel database systems, some mature infrastructures run parallel simulations to handle the skew problem [5], [13], [18], [24]. The primary technique is to perform dynamic repartition when a load imbalance problem exists. Our repartition method borrows some ideas from those systems. Different from these works, however, we apply the repartition idea to the MapReduce implementation for supercomputing systems first. Moreover, we integrate the repartition with the combiner optimizations in the MapReduce workflow.

9 Conclusion

In this paper, we present Mimir, a MapReduce over MPI framework, including a pipeline combiner workflow, a new dynamic repartition method, and a strategy for splitting single superkeys across processes. The three optimizations balance the memory usage for highly skewed datasets up to a memory balance ratio of 1 (i.e., full balancing of data) and reduce the execution time up to 5 times for a diverse set of case studies using the *word count*, *octree clustering*, and *join* benchmarks. At the same time, our framework uses less or equal memory compared with existing state-of-the-art MapReduce over MPI implementations such as MR-MPI. Mimir is highly scalable, scaling to at least 3,072 processes on Tianhe-2. To the best of our knowledge, this is the first work to handle data skew problems in MapReduce over MPI for large-scale supercomputing systems. Our method has been integrated into the Mimir code, which can be downloaded at https://github.com/TauferLab/Mimir.git.

Acknowledgments

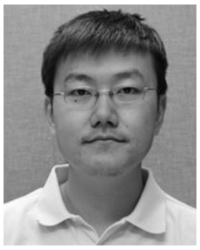
Yanfei Guo and Pavan Balaji were supported by the U.S. Department of Energy, Office of Science, under Contract number DE-AC02-06CH11357. Boyu Zhang, Pietro Cicotti, Tao Gao, and Michela Taufer were supported by US National Science Foundation Grants #1318445, #1318417, and #1841758. Tao Gao and Yutong Lu were supported by the National Key R&D Project in China 2016YFB1000302, the National Natural Science Foundation of China U1611261 and NSFC61402503, and the program for Guangdong Introducing Innovative and Entrepreneurial Teams (2016ZT06D211). Tao Gao was also supported by the China Scholarship Council. Part of the research in this article used resources of the National Supercomputer Center in Guangzhou, China. XSEDE resources, supported by US National Science Foundation grant ACI-1053575, were used to obtain some other performance data.

References

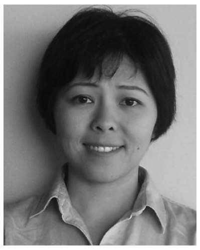
[1] Apahce Hadoop. [Online]. Available: http://hadoop.apache.org/
[2] IBM BG/Q Architecture. [Online]. Available: https://www.alcf.anl.gov/files/IBM_BGQ_Architecture_0.pdf
[3] S. Acharya, P. B. Gibbons, and V. Poosala, “Congressional samples for approximate answering of group-by queries,” *ACM SIGMOD Rec.*, vol. 29, pp. 487–498, 2000.
[4] L. A. Adamic and B. A. Huberman, “Zipf’s law and the internet,” *Glottometrics*, vol. 3, no. 1, pp. 143–150, 2002.
[5] M. Agarwal *et al.*, “Automate: Enabling autonomic applications on the grid,” in *Proc. Autonomic Comput. Workshop*, 2003, pp. 48–57.
[6] G. Ananthanarayanan *et al.*, “Reining in the outliers in MapReduce clusters using Mantri,” in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, vol. 10, 2010, Art. no. 24.
[7] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, “A comparison of join algorithms for log processing in MapReduce,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 975–986.
[8] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, “Scaling spark on HPC systems,” in *Proc. 25th ACM Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2016, pp. 97–110.
[9] Q. Chen, C. Liu, and Z. Xiao, “Improving MapReduce performance using smart speculative execution strategy,” *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 954–967, Apr. 2014.
[10] Q. Chen, J. Yao, and Z. Xiao, “LIBRA: Lightweight data skew mitigation in MapReduce,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 9, pp. 2520–2533, Sep. 2015.
[11] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[12] A. Deshpande, Z. Ives, and V. Raman, “Adaptive query processing,” *Found. Trends® Databases*, vol. 1, no. 1, pp. 1–140, 2007.
[13] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, “Zoltan data management service for parallel dynamic applications,” *Comput. Sci. Eng.*, vol. 4, no. 2, pp. 90–97, 2002.
[14] T. Estrada, B. Zhang, P. Cicotti, R. S. Armen, and M. Taufer, “A scalable and accurate method for classifying protein-ligand binding geometries using a MapReduce approach,” *Comput. Biol. Med.*, vol. 42, no. 7, pp. 758–771, 2012.
[15] T. Gao *et al.*, “Mimir: Memory-efficient and scalable MapReduce for large supercomputing systems,” in *Proc. 31th IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 1098–1108.
[16] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, “Load balancing in MapReduce based on scalable cardinality estimates,” in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 522–533.
[17] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, “LEEN: Locality/fairness-aware key partitioning for MapReduce in the cloud,” in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 17–24.
[18] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis, “Mizan: A system for dynamic load balancing in large-scale graph processing,” in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 169–182.
[19] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, “Skew-resistant parallel processing of feature-extracting scientific user-defined functions,” in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 75–86.
[20] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, “SkewTune: Mitigating skew in MapReduce applications,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 25–36.
[21] Y. Le, J. Liu, F. Ergun, and D. Wang, “Online load balancing for MapReduce with skewed data input,” in *Proc. IEEE INFOCOM*, 2014, pp. 2004–2012.
[22] X.-K. Liao *et al.*, “High performance interconnect network for Tianhe system,” *J. Comput. Sci. Technol.*, vol. 30, no. 2, pp. 259–272, 2015.
[23] X. Lu, M. W. U. Rahman, N. Islam, D. Shankar, and D. K. Panda, “Accelerating spark with RDMA for big data processing: Early experiences,” in *Proc. IEEE 22nd Annu. Symp. High-Perform. Interconnects*, 2014, pp. 9–16.
[24] L. Oliker and R. Biswas, “PLUM: Parallel load balancing for adaptive unstructured meshes,” *J. Parallel Distrib. Comput.*, vol. 52, no. 2, pp. 150–177, 1998.
[25] S. J. Plimpton and K. D. Devine, “MapReduce in MPI for large-scale graph algorithms,” *Parallel Comput.*, vol. 37, no. 9, pp. 610–632, 2011.
[26] A. Shatdal and J. F. Naughton, “Adaptive parallel aggregation algorithms,” *ACM SIGMOD Rec.*, vol. 24, pp. 104–114, 1995.
[27] S.-J. Sul and A. Tovchigrechko, “Parallelizing BLAST and SOM algorithms with MapReduce-MPI library,” in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Workshops Phd Forum*, 2011, pp. 481–489.
[28] The Ohio State University. [Online]. Available: http://hibd.cse.ohio-state.edu/
[29] Y. Wang, R. Goldstone, W. Yu, and T. Wang, “Characterization and optimization of memory-resident MapReduce on HPC systems,” in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 799–808.
[30] M. Wasi-ur Rahman *et al.*, “High-performance RDMA-based design of hadoop MapReduce over InfiniBand,” in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process. Workshops PhD Forum*, 2013, pp. 1908–1917.
[31] T. White, *Hadoop: The Definitive Guide*. Sebastopol, CA, USA: O’Reilly Media, Inc., 2012.
[32] M. Xie, Y. Lu, K. Wang, L. Liu, H. Cao, and X. Yang, “Tianhe-1A interconnect and message-passing services,” *IEEE Micro*, vol. 32, no. 1, pp. 8–20, Jan./Feb. 2012.
[33] W. Xu *et al.*, “Hybrid hierarchy storage system in MilkyWay-2 supercomputer,” *Front. Comput. Sci.*, vol. 8, no. 3, pp. 367–377, 2014.
[34] Y. Xu and P. Kostamaa, “Efficient outer join data skew handling in parallel DBMS,” *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1390–1396, 2009.
[35] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, “Handling data skew in parallel joins in shared-nothing systems,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1043–1052.

[36] X. Yang, N. Liu, B. Feng, X.-H. Sun, and S. Zhou, “PortHadoop: Support direct HPC data processing in Hadoop,” in *Proc. IEEE Int. Conf. Big Data*, 2015, pp. 223–232.
[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.
[38] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, “Improving MapReduce performance in heterogeneous environments,” in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, Art. no. 7.

Tao Gao (Student Member, IEEE) is working toward the PhD degree at the National University of Defense Technology, Changsha, China and is a visiting scholar with the University of Delaware. His research interests include big data processing and high-performance computing.



Yanfei Guo (Member, IEEE) is an assistant computer scientist with Argonne National Laboratory. His research interests include cloud computing, big data processing and MapReduce, and high-performance computing.



Boyu Zhang is a software engineer with Microsoft. Her research interests include scalable big data analytics, clustering and classification methods, performance analysis and optimization, and scientific applications.

Pietro Cicotti (Member, IEEE) is a senior architect with NVIDIA, where he is a member of the TensorRT Team. His research interests include architecture, systems design, optimizations, and emerging technologies for exascale, scientific computing, and machine learning.

Yutong Lu (Member, IEEE) is the director of the National Supercomputing Center in Guangzhou, China. She is also a professor with the School of Computer Science, Sun Yat-sen University, as well as with the National University of Defense Technology. Her research interests include parallel operating systems, high-speed communication, global file systems, and advanced programming environments.

Pavan Balaji (Senior Member, IEEE) is a computer scientist with Argonne National Laboratory where he leads the Programming Models and Runtime Systems Group. His research interests include parallel programming models and runtime systems for communication and I/O on extreme-scale supercomputing systems, modern system architecture, cloud computing systems, data-intensive computing, and big data sciences.

Michela Taufer (Senior Member, IEEE) holds the Jack Dongarra professorship in high performance computing with the Department of Electrical Engineering and Computer Science, University of Tennessee Knoxville. Her research interests include high-performance computing, including scientific applications scientific applications, scheduling and reproducibility challenges, and big data analytics.