On the Power of Combiner Optimizations in MapReduce over MPI Workflows

Tao Gao,[a,b] Yanfei Guo,[c] Boyu Zhang,[b] Pietro Cicotti,[f]
Yutong Lu,[d,e] Pavan Balaji,[c] and Michela Taufer[b]

[a]National University of Defense Technology
[b]University of Delaware
[c]Argonne National Laboratory
[d]Sun Yat-sen University
[e]National Supercomputing Center in Guangzhou
[f]San Diego Supercomputer Center

***Abstract*—Analyzing large volumes of data is becoming more and more important in various scientific computing domains. MapReduce over MPI frameworks are an appealing solution to enable scalable big data analytics on supercomputing systems. These systems can further leverage features of MapReduce applications by merging $\langle key/value \rangle$ pairs before the reduce function in combiner optimizations. In this paper, we propose a pipeline combiner workflow and integrate it into Mimir, a cutting-edge implementation of MapReduce over MPI. Our results with real datasets on the Tianhe-2 supercomputer prove that our pipeline combiner workflow can reduce memory usage up to 51% and improve the overall performance up to 61%.**

Keywords: MapReduce over MPI, Combiner Optimization, Memory-efficient, Data Analytics

I. Introduction

Analyzing large volumes of data—big data analytics—is becoming increasingly important in various scientific computing domains such as social sciences, genomics, and pharmaceutical sciences. As data become larger and larger, traditional MapReduce frameworks such as Spark [1] and Hadoop [2] fail to provide scalable solutions for big data analytics. At the same time, supercomputers and high-performance computing (HPC) systems are gaining the attention of the big data community and aim to become the next platform for big data analytics.

Implementations of MapReduce over MPI such as MapReduce-MPI [3], [4] and Mimir [5] have been shown to be scalable MapReduce solutions beyond the scales already supported by Spark or Hadoop. While these MapReduce over MPI implementations do provide scalability, their performance can be further increased by integrating features that leverage properties embedded in the datasets of typical MapReduce applications. In particular, in a large number of MapReduce applications the reduction function is both associative and commutative. Examples of these applications range from searches of more frequent tags in social sciences' tweets or more frequent drug structures in drug design datasets to counting words in large datasets such as Wikipedia.

From an implementation point of view these MapReduce applications support merging $\langle key/value \rangle$ pairs before the reduce function. The merging process is called `combiner` optimization. Combiner optimizations have been used to reduce the amount of data transfer between map and reduce phases in, for example, Spark. When moving to supercomputers, combiner optimizations can reduce memory requirements in systems that have both limited local memory and shared file systems. The implementation of a simple concept such as merging $\langle key/value \rangle$ pairs on HPC systems, however, requires understanding the HPC system's memory hierarchy as well as generating new data structures and tuning the overall MapReduce over MPI workflow.

In this paper we define both new abstractions and implementations of combiner optimizations for HPC systems, starting from a naive combiner workflow based on compressions to a more advanced pipeline combiner workflow based on the effective interleaving of *map* and *shuffle* phases. Our work is driven by the need to minimize memory requirements and reduce data transfer over the network during the MapReduce job's execution. We integrate our pipeline combiner workflow into Mimir because this framework is considered a cutting-edge MapReduce over MPI frameworks (see [5], [6]). Our augmented version of Mimir uses memory more efficiently compared with previous implementations of the same framework. The contributions of this paper are as follows:

- We design a set of combiner optimizations for HPC systems and integrate them in Mimir, a cutting-edge MapReduce over MPI framework.
- We compare and contrast the augmented MapReduce workflow with other MapReduce workflows for datasets in the social sciences (e.g., Wikipedia datasets) and pharmaceutical sciences (e.g., for classifying drug metadata from protein-ligand docking simulations).

We present results on the Tianhe-2 supercomputer. Our results show how our pipeline combiner workflow not only reduces the memory footprint (i.e., up to 59% for the Wikipedia datasets and up to 20% for the pharmaceutical datasets) but also improves performance for large-scale datasets (i.e., up to 34% for the Wikipedia datasets and up to 15% for the pharmaceutical datasets).

II. Background on MapReduce and Mimir

In this section, we review the MapReduce programming model [7] and the cutting-edge MapReduce over MPI framework, Mimir [5].

A. MapReduce Programming Model

MapReduce is a programming model intended for data-intensive applications [7] that has proved to be suitable for a wide variety of applications. A MapReduce job usually involves three phases—*map*, *shuffle*, and *reduce*—as shown in Figure 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 $\langle key/multiple\,values \rangle$ (KMV) lists. The *reduce* phase processes the KMV lists with a user-defined reduce callback function and generates the final output. The user needs to implement the map and reduce callback functions, while the MapReduce runtime handles the parallel job execution, shuffle communication, and data movement.

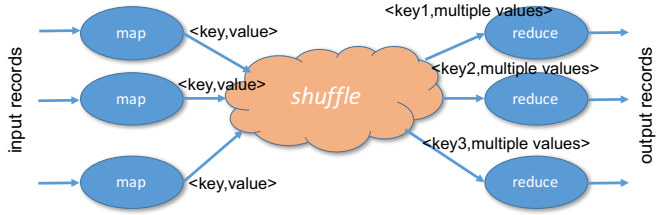


Fig. 1: The map, shuffle, and reduce phases of the MapReduce programming model.

B. Mimir Overview

Mimir is a state-of-the-art MapReduce over MPI framework [5]. As in Hadoop [2] and Spark [1], a user of Mimir defines operations to be used in the `map` and `reduce` phases. MPI processes execute these operations. Mimir pipelines and interleaves the computation and shuffle communication within the map phase operation to minimize unnecessary memory usage, ensuring high performance on supercomputers. Different from the master-worker architecture used in Hadoop[2] and Spark [1], Mimir is designed to be decentralized in order to increase scalability.

In Mimir's map phase shown in Figure 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 operation 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 other 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 into its receive buffer partitions. Once the KVs are in the receive buffer, each process moves the KVs into a KV container (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.

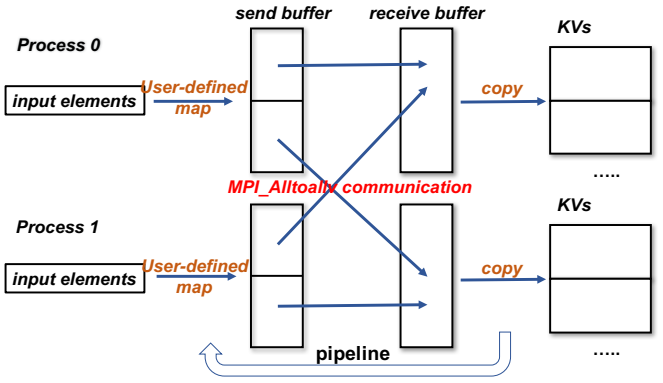


Fig. 2: Map workflow in Mimir with its map phase including both computation and shuffle communication stages.

In Mimir's reduce phase shown in Figure 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 KV-to-KMV 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). 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.

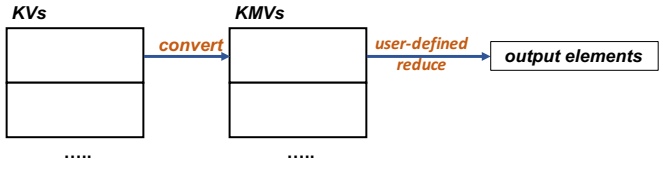


Fig. 3: Workflow of the reduce phase in Mimir with the two-pass algorithm to perform the KV-to-KMV conversion in memory.

III. Pipeline Combiner Workflow

In this section, we present combiner optimizations to minimize the memory usage for those MapReduce applications that

are both associative and commutative and therefore support merging KVs with the same key before the reduce phase.

A. Naive Combiner Workflow

A naive combiner implementation in Mimir [5] applies compressions and partial reductions before and after the shuffle communication, respectively. As shown in Figure 4, the design is driven by the motivation of maximizing the reduction in data that is sent over the network. Nevertheless, the implementation still suffers from delays: the shuffle communication is postponed until the compression has finished. In other words, the map computation stage is not pipelined with the shuffle communication in the combiner implementation. This constraint forces the introduction of an extra intermediate data staging before the shuffle communication.

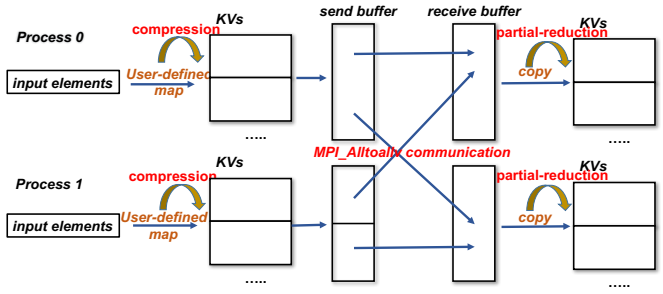


Fig. 4: Workflow of the compression and partial reduction in Mimir.

B. Pipeline Combiner Workflow

We leverage the naive workflow with compression and partial reduction and extend Mimir's workflow to support a pipeline combiner workflow as shown in Figure 5. From an implementation point of view, we extend MapReduce applications by allowing them to set a new combiner callback function. The combiner callback takes two values as input and generates a single value as output. The figure points out how the combiner callback can be applied both before and after each shuffle communication stage of the map phase.

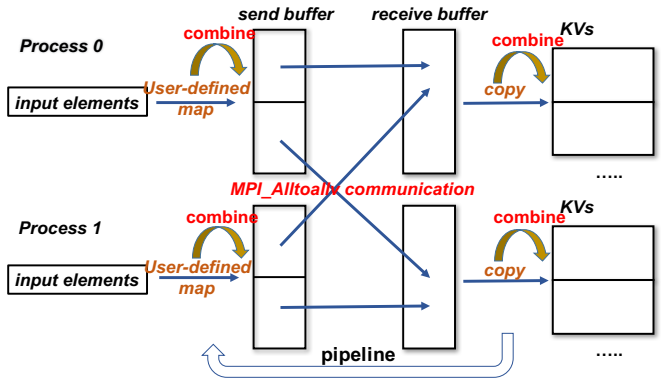


Fig. 5: Combiner workflow in Mimir with its combiner callbacks applied before and after each communication stage.

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 WordCount 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 use another 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.

C. Garbage Management

During the combiner optimization, the length of the combined KV may differ from that of the original KV in the send buffer or in the KVC. For example, the value field is a variable-length string. As a result, the combined KV may create some "holes" (i.e., garbage bytes between different KVs). For each process, we introduce two additional hash buckets to track these holes: one for the hash bucket before the shuffle communication stage and one for the hash bucket after the shuffle communication stage.

If the length of the combined KV is equal to the original KV in the send buffer, then the combined version is stored in the same position as the original KV, and no hole is created.

If the length of the combined KV is less than that of the original KV in the send buffer, then the combined version is stored in the same position as the original KV, and a hole is generated. Otherwise, if the length of the combined KV is greater than that of the original KV in the send buffer, then the combined version is stored in another position, and the bytes of the original KV are marked as a hole. When searching for a location for a new KV, we look for a hole that is large enough to hold the KV first. If such a hole does not exist, then the KV is inserted at the end of the buffer.

Before starting a shuffle communication stage, a garbage collection phase is applied to remove any existing holes. During garbage collection, the process's buffer is scanned from the beginning to the end. If a hole is found, then the KVs after that hole are moved forward to eliminate the hole. For each process, we introduce one hash bucket to track these holes. We note that we can check whether an address is the start address of a hole in $O(1)$ time with a hash bucket.

In the case of the KVC, the combiner implementation is similar to the implementation for the send buffer. To avoid generating too many garbage bytes, we perform garbage collection if the total garbage size is larger than a user-defined value (two pages by default). Since the maximum size of the aggregated garbage bytes is a constant value, the spatial complexity for these hash buckets is $O(1)$.

D. Memory Usage Analysis

To simplify the memory usage analysis, we assume that the lengths of each key and each value are the same. This is a reasonable assumption because the lengths of the keys and the values are often small and approximately the same. Let us assume that n represents the total number of KVs and that u represents the number of unique keys in those KVs. We have p processes to execute the MapReduce tasks. Note that u is always equal to or smaller than n. We further assume that the unique keys are partitioned evenly to all processes.

In the naive combiner workflow of Mimir (shown in Figure 4), only the memory usage of the intermediate data buffers (i.e., KVCs) depends on the dataset size. The worst spatial complexity of KVCs before shuffle communication is $O(p*u)$ because each process may encounter all unique keys; and the best spatial complexity of the KVCs before shuffle communication is $O(u)$. We note that the best situation is rare in real-world datasets. On average, the spatial complexity of KVCs before shuffle communication is $O((p+1)*p/2*u)$. The spatial complexity of KVCs after shuffle communication is $O(u)$. Thus, the overall spatial complexity on average is $O((p+1)*p/2*u+u)$ in the naive combiner workflow.

In the pipeline combiner workflow, 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). Since the number of processes p is large when running on supercomputers, the pipeline workflow uses memory much more efficiently compared with the naive combiner workflow.

IV. Evaluation

In this section, we evaluate the performance of the pipeline combiner workflow for datasets in social sciences (e.g., Wikipidia datasets) and in pharmaceutical sciences (i.e., for classifying drug metadata from protein-ligand docking simulations).

A. Platforms, Settings, Benchmarks, and Datasets

Our tests are performed on the Tianhe-2 high-performance supercomputer located at the National Supercomputer Center in Guangzhou, China. Each compute node has two Intel Xeon E2-2692v2 CPUs (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 [8], and the parallel file system is H2FS [9]. We use MPICH 3.1.3 with a customized GLEX channel on the Tianhe-2 [10].

We use two benchmarks that are diverse in terms of datasets and features: *WordCount* (WC) and *octree clustering* (OC). WC is a single-pass MapReduce application that counts the number of occurrences of each unique word in given input files; it is the base algorithm for analyzing social datasets. We test WC with two kinds of datasets: synthetic-generated datasets, whose words are generated following the Zipf distribution [11], and the Wikipedia dataset from the PUMA benchmark suite [12]. We choose the Zipf distribution with the degree of skew (i.e., α in Zipf distribution) being 1.0 in order to simulate the Wikipedia dataset in synthetic datasets.

OC, on the other hand, is an iterative MapReduce application with multiple MapReduce stages. OC is a clustering algorithm for points in a three-dimensional space. For our tests, we use the MapReduce algorithm described by Estrada et al. [13] for classifying points representing ligand metadata from protein-ligand docking simulations; the dataset we use is from Zhang et al. [14], [15]. We test the performance of our augmented MapReduce workflow with different densities of the dataset. The different densities represent different outcomes of the protein-ligand docking simulations: the higher the density, the more likely a simulation is identifying a single drug conformation of interest. Pragmatically, if the density of the drug metadata dataset is set to x, then the algorithm finds the smallest cube that contains $x*n$ metadata points, where n is the total number of drug conformations generated by the simulations and collected in the dataset.

Our metrics of success are average peak memory usage and average execution time. Peak memory usage is the maximum memory usage at any point in time during the application execution. 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 our experimental platforms. We run one process per core for all tests and repeat the tests at least three times.

B. Single-Node Comparison

In this section, we compare our pipeline combiner workflow with the naive combiner workflow in Mimir on a single node

of the Tianhe-2 supercomputer with 24 processes. For WC, we test with both synthetic datasets and Wikipedia datasets; for OC, we test with the datasets from [14], [15].

The results for the WC benchmark with synthetic datasets are shown in Figure 6. We fix the dataset size to 24 GB (each word is 5 bytes) and vary the number of unique words (i.e., 1 million, 10 million, and 100 million). The average memory usage per process is shown in Figure 6a. As the figure indicates, the pipeline combiner workflow reduces the memory usage for all three datasets. The more diverse the dataset, the more tangible the reduction. The reason is that the memory usage of the native implementation increases much faster compared with the pipeline implementation as the number of unique words increases, as discussed in Section III-D. The reduction of memory usage for the dataset with 100 million unique words is up to 51%. The execution time is shown in Figure 6b. The pipeline workflow reduces the execution time for datasets with 10 million and 100 million unique words. The reduction for the dataset with 100 million unique words is up to 61%.

The results for the WC benchmark with the Wikipedia datasets are shown in Figure 7. We use three different Wikipedia datasets (i.e., 50 GB, 150 GB, and 300 GB) from the PUMA benchmark suite [12]. The number of unique words in the three datasets is about 100 million. Memory usage is compared in Figure 7a, and execution time is compared in Figure 7b. The pipeline workflow reduces memory usage for all three datasets. The memory usage reduction of datasets with 50 GB, 150 GB, and 300 GB is about 22%, 48%, and 59%, respectively. The pipeline workflow slightly slows the execution for small datasets (i.e., 50 GB) because of overheads, but it significantly improves the performance for 150 GB and 300 GB datasets. The performance improvement for 300 GB datasets is up to 34%.

The results for the OC benchmark are shown in Figure 8. We fix the number of points (drug metadata) to 2^{29} and test with different densities of the points (i.e., 1%, 0.1%, and 0.01%). As shown in Figure 8a, the pipeline combiner reduces the peak memory usage 20% for the 0.1% and 0.01% settings. The pipeline workflow also improves the performance for datasets with 0.1% and 0.01% settings up to 15%, as shown in Figure 8b. The peak memory usage and execution time reduction for the 1% setting are limited because the number of unique keys within intermediate datasets in this case is small. The pipeline combiner workflow has more improvement when there are more unique keys in the intermediate datasets.

C. Scalability on Multiple Nodes

In this section, we compare the scalability of our pipeline combiner workflow with the naive combiner workflow in Mimir on the Tianhe-2 supercomputer.

The scalability results for the WC benchmark with synthetic datasets are shown in Figure 9. We fix the number of points per node (24 processes) to 24 GB and the number of unique words per node to 10 million. That is, both the dataset size and number of unique words increase as the process count increases. The peak memory usage per process and execution is shown in Figure 9a and Figure 9b, respectively. As shown in Figure 9a, the memory usage per process of the pipeline workflow remains constant up to 1,536 processes while the memory usage of the naive workflow increases as the number of processes increases. The naive implementation cannot execute with 768 processes or more because of running out of memory. The reason is that the memory usage in the naive workflow is decided by both the number of unique words and the number of processes, as discussed in Section III-D. The combiner workflow also reduces the execution time significantly, as shown in Figure 9b. The improvement for the setting with 384 processes is close to 3 times.

The results for the OC benchmark are shown in Figure 10. We fix the number of points per node (24 processes) to 2^{29} and set the metadata density to 0.01%. The average memory usage per process is shown in Figure 10a. As shown in this figure, the reduction in memory usage is about 20% from 24 processes (single node) to 1,536 processes (multiple nodes). As shown in Figure 10b, the pipeline combiner also reduces the execution time.

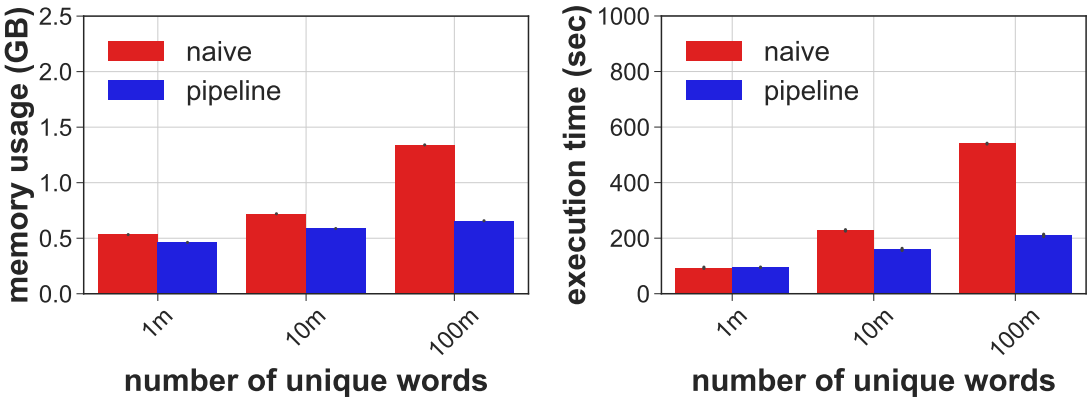
To sum up, the results of our tests provide quantitative evidence that the combiner workflow reduces the memory usage and improves the overall performance for a diverse set of benchmarks and their datasets, especially when the number of unique keys is significantly large in intermediate datasets during the MapReduce workflow execution.

V. Related Work

Combiner optimizations have been applied in most popular MapReduce frameworks, such as Hadoop [2], Spark [16], and MapReduce-MPI [4]. Most of the work seeks to maximize the benefits of combiner optimizations in order to reduce the amount of data sent over the network. Thus, these frameworks do not pipeline the local combination with the shuffle communication, as in our design. While Hadoop has been replaced by Spark in cloud computing, our previous work showed the lack of scalability in Spark [5] and the memory usage challenges when using MapReduce-MPI [5]. Our pipeline design further pushes the performance of the augmented Mimir with our combiner optimizations to outperform the other frameworks in terms of their scalability and memory usage.

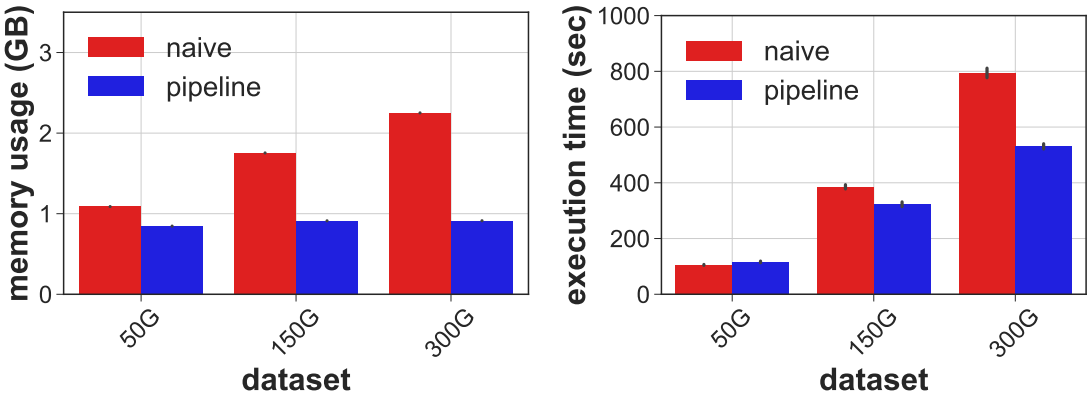
In distributed systems, pipelines have been used in MRO-MPI [17]. The MRO-MPI model has an optimized data exchange policy to overlap map computation and shuffle communication. In this design, the combiner applied at the receive side is also pipelined with the shuffle communication. Our design has significant differences. Specifically, our combiner optimizations are applied at both the send and the receive side; thus, it can reduce the amount of data sent over the network and reduce the memory requirement. In contrast, their design cannot reduce the amount of data sent over the network.

In shared-memory systems, pipeline combiner workflows have been implemented in MapReduce frameworks. For example, in Tiled-MapReduce [18], the pipeline design outperformed other shared-memory implementations, such as



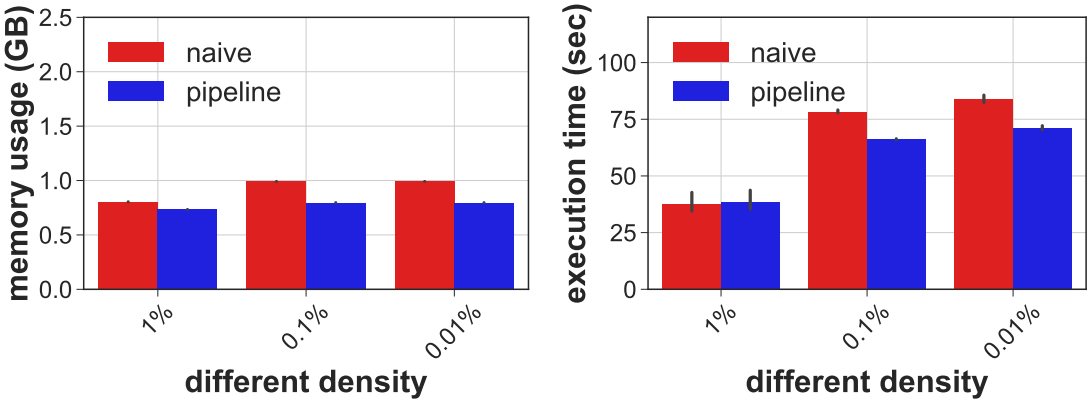
(a) Peak memory usage (per process) (b) Execution time

Fig. 6: Single-node results (24 processes) of our pipeline combiner workflow compared with the naive combiner workflow in Mimir with the WC benchmark and 24 GB synthetic datasets. The number of unique words in these datasets is 1 million (1m), 10 million (10m), and 100 million (100m), respectively.



(a) Peak memory usage (per process) (b) Execution time

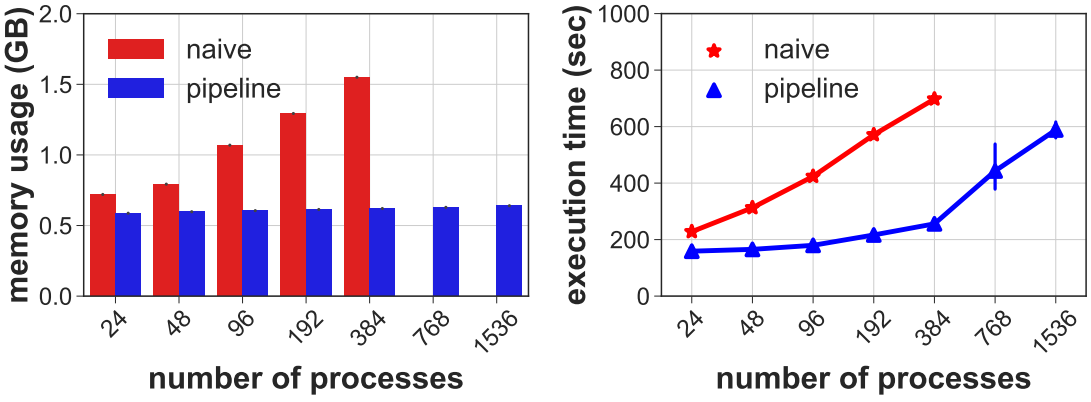
Fig. 7: Single-node results (24 processes) of our pipeline combiner workflow compared with the naive combiner workflow in Mimir with the WC benchmark and 50 GB, 150 GB, and 300 GB Wikipedia datasets from the PUMA benchmark suite.



(a) Peak memory usage (per process) (b) Execution time

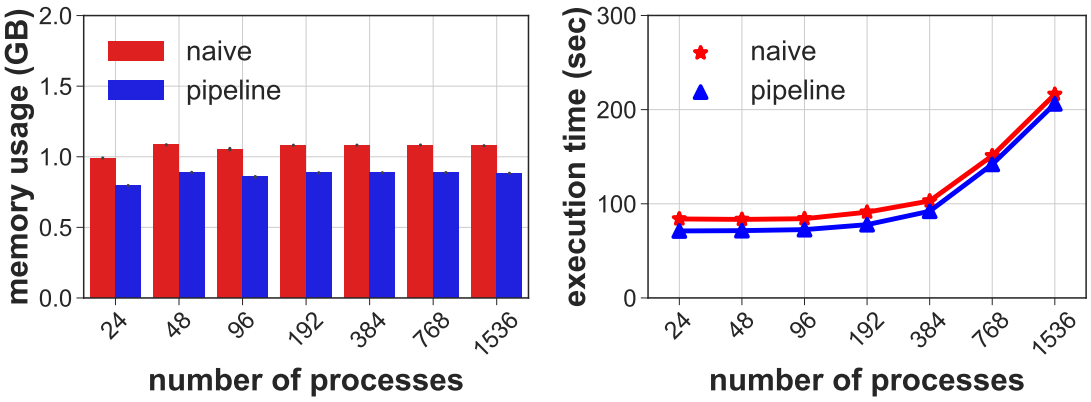
Fig. 8: Single-node results (24 processes) of our pipeline combiner workflow compared with the naive combiner workflow in Mimir with the OC benchmark and different drug metadata densities (i.e., 1%, 0.1%, and 0.01%).

Phoenix [19] and Phonix++ [20]. We note, however, that the Tiled-MapReduce design cannot work on large-scale distributed-memory systems, and thus it suffers from scalability limitations.



(a) Peak memory usage (per process) (b) Execution time

Fig. 9: Scalability results of our pipeline combiner workflow compared with the naive combiner workflow in Mimir with the WC benchmark and synthetic datasets.



(a) Peak memory usage (per process) (b) Execution time

Fig. 10: Scalability results of our pipeline combiner workflow compared with the naive combiner workflow in Mimir with the OC benchmark.

VI. Conclusion

In this paper, we propose a pipeline combiner workflow that leverages features of MapReduce applications such as being both associative and commutative, in order to further increase performance and memory usage when these application are executed on MapReduce over MPI frameworks. We integrate our pipeline combiner workflow into Mimir, a cutting-edge MapReduce over MPI framework. Results on the large-scale supercomputer Tianhe-2 demonstrate that our proposed design outperforms previous designs by reducing the memory requirement up to 59% and execution time up to 61%. Our design also presents better scalability compared with previous designs. These methods have been merged into the Mimir code; the code can be downloaded at https://github.com/TauferLab/Mimir.git.

Acknowledgment

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 NSF grants #1318445 and #1318417. 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. The research in this paper used resources of the National Supercomputer Center in Guangzhou, China.

License

DOE Public Access Plan. http://energy.gov/downloads/doe-public-access-plan.

REFERENCES

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," *HotCloud*, vol. 10, pp. 10–10, 2010.

[2] "Apahce Hadoop," http://hadoop.apache.org/.

[3] S.-J. Sul and A. Tovchigrechko, "Parallelizing BLAST and SOM Algorithms with MapReduce-MPI Library," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*, 2011, pp. 481–489.

[4] S. J. Plimpton and K. D. Devine, "MapReduce in MPI for Large-Scale Graph Algorithms," *Journal of Parallel Computing*, vol. 37, no. 9, pp. 610–632, 2011.

[5] 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," in *Proceedings of the 31th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[6] T. Gao, Y. Guo, Y. Wei, B. Wang, Y. Lu, P. Cicotti, P. Balaji, and M. Taufer., "Bloomfish: A Highly Scalable Distributed K-mer Counting Framework," in *Proceedings of the 23nd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2017.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[8] X.-K. Liao, Z.-B. Pang, K.-F. Wang, L. Y., M. Xie, J. Xia, D. D.-Z., and G. Suo, "High Performance Interconnect Network for Tianhe System," *Journal of Computer Science and Technology*, vol. 30, no. 2, pp. 259–272, 2015.

[9] W. Xu, Y. Lu, Q. Li, E. Zhou, Z. Song, Y. Dong, W. Zhang, D. Wei, X. Zhang, H. Chen, J. Xing, and Y. Yuan, "Hybrid Hierarchy Storage System in MilkyWay-2 Supercomputer," *Frontiers of Computer Science*, vol. 8, no. 3, pp. 367–377, 2014.

[10] 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, 2012.

[11] L. A. Adamic and B. A. Huberman, "Zipf's Law and the Internet," *Journal of Glottometrics*, vol. 3, no. 1, pp. 143–150, 2002.

[12] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "PUMA: Purdue MapReduce Benchmarks Suite," Department of Engineering and Computer Engineering, Purdue University, Tech. Rep., 2012.

[13] 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," *Journal of Computers in Biology and Medicine*, vol. 42, no. 7, pp. 758–771, 2012.

[14] B. Zhang, T. Estrada, P. Cicotti, and M. Taufer, "On Efficiently Capturing Scientific Properties in Distributed Big Data without Moving the Data: A case Study in Distributed Structural Biology using MapReduce," in *Proceedings of the 2013 IEEE 16th International Conference on Computational Science and Engineering (CSE)*, 2013, pp. 117–124.

[15] B. Zhang, T. Estrada, P. Cicotti, P. Balaji, and M. Taufer, "Enabling Scalable and Accurate Clustering of Distributed Ligand Geometries on Supercomputers," *Journal of Parallel Computing*, vol. 63, pp. 38–60, 2017.

[16] "Apahce Spark," http://spark.apache.org/.

[17] H. Mohamed and S. Marchand-Maillet, "MRO-MPI: MapReduce Overlapping Using MPI and an Optimized Data Exchange Policy," *Journal of Parallel Computing*, vol. 39, no. 12, pp. 851–866, 2013.

[18] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010, pp. 523–534.

[19] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a Large-Scale Shared-Memory System," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2009, pp. 198–207.

[20] J. Talbot, R. M. Yoo, and C. Kozyrakis, "Phoenix++: Modular MapReduce for Shared-memory Systems," in *Proceedings of the Second International Workshop on MapReduce and Its Applications*, 2011, pp. 9–16.