Mimir: Memory-Efficient and Scalable MapReduce for Large Supercomputing Systems

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

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

***Abstract*—In this paper we present *Mimir*, a new implementation of MapReduce over MPI. Mimir inherits the core principles of existing MapReduce frameworks, such as MR-MPI, while redesigning the execution model to incorporate a number of sophisticated optimization techniques that achieve similar or better performance with significant reduction in the amount of memory used. Consequently, Mimir allows significantly larger problems to be executed in memory, achieving large performance gains. We evaluate Mimir with three benchmarks on two high-end platforms to demonstrate its superiority compared with that of other frameworks.**

Keywords: High-performance computing; Data analytics; MapReduce; Memory efficiency; Performance and scalability

I. 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. Analyzing and understanding large volumes of data are becoming increasingly important in various scientific computing domains, often as a way to find anomalies in data, although other uses are being actively investigated as well. Big data analytics has recently grown into a popular catch-all phrase that encompasses various analytics models, methods, and tools applicable to large volumes of data. MapReduce is a programming paradigm within this broad domain that—loosely speaking—describes one methodology for analyzing such large volumes of data.

We note that big data analytics and MapReduce are not inventions of the scientific computing community, although several ad hoc tools with similar characteristics have existed for several decades in this community. These are generally considered borrowed concepts from the broader data analytics community [10] that has also been responsible for developing some of the most popular implementations of MapReduce, such as Hadoop [27] and Spark [30]. While these tools provide an excellent platform for analyzing various forms of data, the hardware/software architectures that they target (i.e., generally Linux-based workstation clusters) are often different from that which scientific computing applications target (i.e., large supercomputing facilities).

While commodity clusters and supercomputing platforms might seem similar, they have subtle differences that are important to understand. 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 [3] use specialized lightweight operating systems that do not provide the same capabilities as what a traditional operating system such as Linux or Windows might.

Researchers have attempted to bridge the gap between the broader data analytics tools and scientific computing in a number of ways. These attempts can be divided into four categories: (1) deployment of popular big data processing frameworks on high-performance computers [24], [17], [26], [9]; (2) extension to the MPI [5] interface to support $\langle key, value \rangle$ communication [16]; (3) building of MapReduce-like libraries to support in situ data processing on supercomputing systems [25]; and (4) building of an implementation of MapReduce on top of MPI [21]. Of these, MapReduce implementations over MPI—particularly MR-MPI [21]—have gained the most traction for two reasons: they provide C/C++ interfaces that are more convenient to integrate with scientific applications compared with Java, Scala, or Python interfaces, which are often unsupported on some large supercomputers; and they do not require any extensions to the MPI interface.

MR-MPI has taken a significant first step in bridging the gap between data analytics and scientific computing. It embodies the core principles of MapReduce, including scalability to large systems, in-memory processing where possible, and spillover to the I/O subsystem for handling large datasets; and it does so while allowing scientific applications to easily and efficiently take advantage of the MapReduce paradigm [31], [22]. Yet despite its success, the original MR-MPI implementation still suffers from several shortcomings. One shortcoming is its inability to handle system faults: we addressed this shortcoming in our previous work [12]. Another significant shortcoming is its simple memory management. Specifically,

MR-MPI uses a model based on fixed-size "pages": MR-MPI pages are static memory buffers that are allocated at the start of each MapReduce phase and used throughout this phase. As long as the application dataset can fit in these pages, the data processing is in memory. But as soon as the application dataset is larger than what fits in these pages, MR-MPI spills over the data into the I/O subsystem. While this model is functionally correct, it leads to a tremendous loss in performance.

Figure 1 illustrates this point with the *WordCount* benchmark on a single compute node of the Comet cluster at the San Diego Supercomputing Center (cluster details are presented in Section IV). We note that while MR-MPI provides the necessary functionality for this computation, it experiences significant slowdown in performance for datasets larger than 4 GB, even though the node itself contains 128 GB of memory. Consequently, increasing the dataset size from 4 GB to 64 GB results in nearly three orders of magnitude degradation in performance.

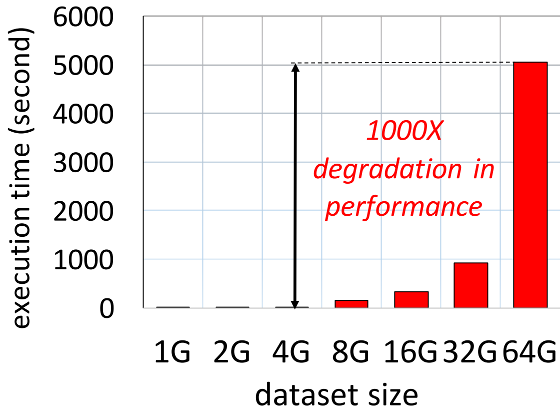


Fig. 1: Single-node execution time of *WordCount* with MR-MPI on Comet.

The goal of the work presented here is to overcome such inefficiencies and design a memory-efficient MapReduce library for supercomputing systems. To this end, we present a new MapReduce implementation over MPI, called *Mimir*. Mimir inherits the core principles of MR-MPI while redesigning the execution model to incorporate a number of sophisticated optimization techniques that significantly reduce the amount of memory used. Our experiments demonstrate that for problem sizes where MR-MPI can execute in memory, Mimir achieves equal or better performance than does MR-MPI. At the same time, Mimir allows users to run significantly larger problems in memory, compared with MR-MPI, thus achieving significantly better performance for such problems.

The rest of this paper is organized as follows. We provide a brief background of MapReduce and MR-MPI in Section II. In Section III, we introduce the design of Mimir and present experimental results demonstrating its performance in Section IV. Other research related to our paper is presented in Section V. We finally draw our conclusions in Section VI.

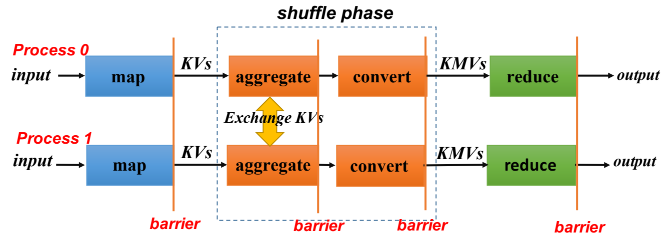


Fig. 2: The *map*, *shuffle*, and *reduce* phases in MR-MPI.

II. BACKGROUND

In this section, we provide a high-level overview of the MapReduce programming model and the MR-MPI implementation of MapReduce.

A. MapReduce Programming Model

MapReduce is a programming model intended for data-intensive applications [10] that has proved to be suitable for a wide variety of applications. A MapReduce job usually involves three phases: *map*, *shuffle*, and *reduce*. 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 communication that distributes the intermediate KV pairs across all processes. In this phase KV pairs with the same key are also merged and stored in $\langle key, \langle value1, value2... \rangle \rangle$ (KMV) lists. The *reduce* phase processes the KMV lists with a user-defined reduce callback function and generates the final output. A global barrier between each phase ensures correctness. The user needs to implement the map and reduce callback functions, while the MapReduce runtime handles the parallel job execution, communication, and data movement.

Several successful implementations of the MapReduce model exist, such as Hadoop [1] and Spark [30]. These frameworks seek to provide a holistic solution that includes the MapReduce engine, job scheduler, and distributed file system. However, large supercomputing facilities usually have their own job scheduler and parallel file system, thus making deployment of these existing MapReduce frameworks in such facilities impractical.

B. MapReduce-MPI (MR-MPI)

MR-MPI is a MapReduce implementation on top of MPI that 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. Figure 2 shows the workflow of MR-MPI. The `aggregate` phase handles the all-to-all movement of data between processes. Within 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.

Frequent allocation and deallocation of memory buffers with different sizes can result in memory fragmentation. Unfortunately, some supercomputing systems, such as the IBM BG/Q, use lightweight kernels with a simple memory manager that does not handle such memory fragmentation [7]. In order to avoid memory fragmentation, MR-MPI uses a fixed-size buffer structure called *page* to store the intermediate data. An MR-MPI *page* is simply a large memory buffer and has no relationship to operating system pages. By default, the size of a *page* is 64 MB, although it is configurable by the user. Generally, a user needs to set a larger *page* size in order to use the system memory more effectively. For each MapReduce phase, MR-MPI tries to allocate all the *pages* it needs at once. The minimum number of pages needed by the *map*, *aggregate*, *convert*, and *reduce* phases is 1, 7, 4, and 3, respectively.

The coarse-grained memory allocation in MR-MPI leads to an efficiency problem: not all the allocated *pages* are fully utilized. For some MapReduce jobs, the size of intermediate data decreases as the data passes through different phases. For example, during the conversion from KVs to KMVs, the values with the same key are grouped together, and the duplicate keys are dropped. If all KVs fit in one *page*, the merged KMVs will be smaller than the *page* size, and thus the buffer storing the KMVs will be underutilized. While some *pages* still have space, other *pages* may already be full. When a *page* is full, MR-MPI writes the contents of the *page* to the I/O subsystem (referred to as I/O spillover in MapReduce frameworks). MR-MPI supports three out-of-core writing settings: (1) always write intermediate data to disk; (2) write intermediate data to disk only when the data is larger than a single *page*; and (3) report an error and terminate execution if the intermediate data is larger than a single page size. Because supercomputing systems generally do not have local disks, the I/O subsystem to which the *page* can be written is often the global parallel file system. This makes the I/O spillover expensive.

Aside from the inefficient use of memory buffer space, MR-MPI suffers from redundant memory buffers and unnecessary memory copies. Figure 3 shows the seven *pages* used in the `aggregate` phase. The first step in `aggregate` is to partition the KVs using the hash function. MR-MPI determines to which process each KV should be sent and the total size of the data to be sent to each process. MR-MPI uses two temporary buffers to store structures related to partitioning of data. After partitioning, MR-MPI copies the KVs from `map`'s output buffer to the send buffer and uses `MPI_Alltoallv` to exchange the data with all processes. The received KVs are then stored in the receive buffer. Because the partitioning of KVs is not guaranteed to be fully balanced, some processes may receive significantly more data than others. MR-MPI

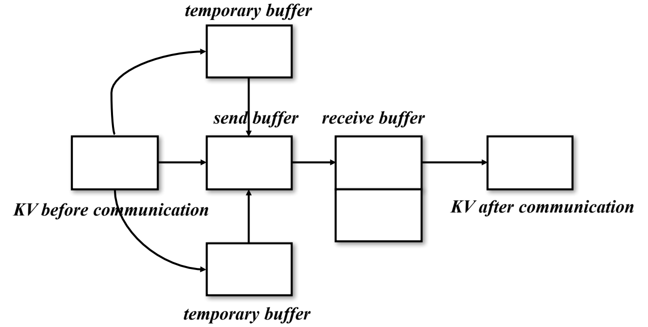


Fig. 3: Memory usage in *aggregate* phase.

allocates two *pages* for the receive buffer to prevent buffer overflow due to partitioning skew. The `aggregate` phase copies the received KVs to the input buffer of the succeeding `convert` phase. Overall, the `aggregate` uses seven *pages*. However, at least two of them—the `map`'s output buffer and the `convert`'s input buffer—are redundant. They can be avoided if the preceding `map` phase uses the send buffer as the output buffer and the succeeding `convert` phase uses the receive buffer as the input buffer. Inserting the output of the preceding `map` directly into send buffer also can reduce the use of temporary buffers by partitioning the KVs directly. A more sophisticated workflow can also eliminate the possibility of receive buffer overflow, thus reducing the size of the receive buffer by half.

III. Design of Mimir

The primary design goal of Mimir is to allow for a memory-efficient MapReduce implementation over MPI. The idea is to have Mimir achieve the same performance as MR-MPI for problem sizes where MR-MPI can execute in memory, while at the same time allowing users to run significantly larger problems in memory, compared with MR-MPI, thus achieving substantial improvement in performance for such problems.

Mimir's execution model offers three classes of improvements that allow it to achieve significant memory efficiency. The first two classes (Sections III-A and III-B) are "core" optimizations; that is, they are an essential part of the Mimir design and are independent of the user application. The third class (Section III-C) is "optional" optimizations; that is, the application needs to explicitly ask for these optimizations depending on the kind of dataset and the kind of processing being done on the data.

Mimir inherits the concepts of KVs and KMVs from MR-MPI. However, it introduces two new objects, called KV containers (KVCs) and KMV containers (KMVCs), to help manage KVs and KMVs. The 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. 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

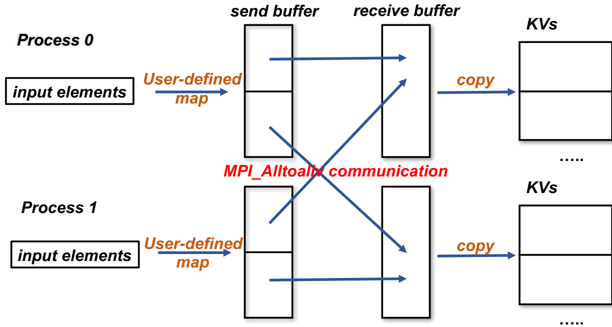


Fig. 4: Workflow of `map` and `aggregate` phases in Mimir.

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.

A. *Mimir Workflow Phases (Core Optimizations)*

Like MR-MPI, Mimir's MapReduce workflow consists of four phases: `map`, `aggregate`, `convert`, and `reduce`. A key difference from MR-MPI, however, is that in Mimir the `aggregate` and `convert` phases are implicit; that is, the user does not explicitly start these phases. This design offers two advantages. First, it breaks the global synchronization between the `map` and `aggregate` phases and between the `convert` and `reduce` phases. Thus, Mimir has more flexibility to determine when the intermediate data should be sent and merged. It also has the flexibility to pipeline these phases to minimize unnecessary memory usage. We still retain the global synchronization between the `map` and `reduce` phases, which is required by the MapReduce programming model. Second, it enables and encourages buffer sharing between the `map` and `aggregate` phases, which can help reduce memory requirements.

Figure 4 shows the workflow within the `map` and `aggregate` phases. Each MPI process has a send buffer and a receive buffer. The send buffer of the MPI process is divided into p equal-sized partitions, where p is the number of processes in the MapReduce job. 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 by using a hash function based on the key. The aim is to ensure that KVs with the same key are sent to the same process. Users can provide alternative hash functions that suit their needs, but the workflow stays the same.

If a partition in the send buffer is full, we temporarily suspend the `map` phase and switch to the `aggregate` phase. In this phase, 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

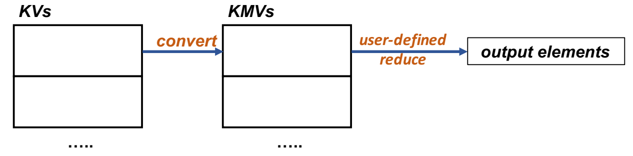


Fig. 5: Workflow of `convert` and `reduce` phases in Mimir.

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 `aggregate` phase completes, and the suspended `map` phase resumes. In this way, the `map` and `aggregate` phases are interleaved, allowing them to process large volumes of input data without correspondingly increasing the memory usage.

In the core design of Mimir, two user-defined callback functions must be implemented by the application: the map and reduce callback functions. In Section III-C we will introduce additional optional callback functions that user applications can implement for additional performance improvements.

Mimir supports three different types of input data sources: files from disk, KVs from previous MapReduce operations for multistage jobs or iterative MapReduce jobs, and sources other than MapReduce jobs (e.g., in situ analytics workflows).

Figure 5 shows the workflow of the `convert` and `reduce` phases in Mimir. In the `convert` phase, the input KVs are stored in a KVC that is generated by the `aggregate` phase. The `convert` phase converts these KVs into KMVs and stores them in a KMVC. We adopt a two-pass algorithm to perform the KV-KMV conversion. 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 KMVC. In the second pass, the KVs are converted into KMVs by inserting them into the corresponding position in the KMVC. When all the KVs are converted to KMVs, the `convert` phase is complete. We then switch to the `reduce` phase and call the user-defined reduce callback function on the KMVs. We note that unlike the `map` and `aggregate` phases, the `convert` and `reduce` phases cannot be interleaved.

B. *Memory Management in Mimir (Core Optimizations)*

Mimir uses two types of memory buffers: data buffers for storing intermediate KVs and KMVs, and communication buffers. Unlike MR-MPI, which statically allocates two large data buffers for the KVs and KMVs, Mimir allows data buffers to be dynamically allocated as the sizes of KVs and KMVs grow. We create KVCs and KMVCs to manage the data buffers.

Mimir creates two communication buffers: a send buffer and a receive buffer. These buffers are statically allocated with the same size. The size is configurable by the user and does not need to be equal to the size of a data buffer. As mentioned in Section III-A, the send buffer is equally partitioned for each process, and the user-defined map function inserts partitioned KVs directly into the send buffer: there is no additional data copying from a map buffer to a send buffer. Thus, unlike MR-

MPI, we no longer need a temporary buffer to function as a staging area for partitioning the KVs. An unexpected side benefit of this design is that it ensures that the size of received data is never larger than the send buffer, even when the KV partitioning is highly unbalanced. As a result, Mimir never needs to allocate a receive buffer that is larger than the send buffer.

C. Mimir Workflow Hints (Optional Optimizations)

This section describes the "optional" optimizations in Mimir. That is, these optimizations are not automatic and need to be explicitly requested for by the application. The reason that these optimizations cannot be automatically enabled is that they assume certain characteristics in the dataset and the computation. If the dataset and computation do not have those characteristics, the result of the computation can be invalid or undefined. Therefore, we provide these capabilities in Mimir but ask that the user explicitly enable them after (manually) verifying that the dataset and computation follow these requirements. These optimizations can be classified into two categories.

The first category is for "advanced functionality." As mentioned in Section III-A, Mimir requires two mandatory user-defined callback functions to be implemented by the user application: the map and reduce callback functions. In this first category of optimizations, the user application can implement additional callbacks that give the user more fine-grained control of the data processing and movement. The "partial reduction" (Section III-C1) and "KV compression" (Section III-C2) optimizations come under this category.

The second category is for "hints," where the user essentially just gives a hint to the Mimir runtime with respect to certain properties of the dataset being processed. There is no change to either the dataset or the computation on the dataset by the application—the application is simply telling Mimir whether the dataset has certain properties or not. The "KV-hint" (Section III-C3) optimization comes under this category.

1) Partial-Reduction: As mentioned in Section III-A, the basic Mimir workflow performs the `convert` and `reduce` phases in a noninterleaved manner. This requires potentially a large amount of memory to hold all the intermediate KMVs in the `convert` phase before `reduce` starts to consume them. While this model ensures correctness for reduce functions, it is conservative. For some jobs, this model leaves unexploited some properties in the dataset, such as "partial-reduce invariance," which is essentially a combination of commutativity and associativity in the reduction operations. With partial-reduce invariance, merging the reduce output in multiple steps, each step processing a partial block of intermediate data, does not affect the overall correctness of the results. An example of such a job is *WordCount*. For these types of MapReduce jobs, the reduce can start as soon as some of the intermediate KVs are available, without waiting for the KVs to be converted to KMVs. This design allows us to perform reductions even when the available memory is less than that required to store all KMVs.

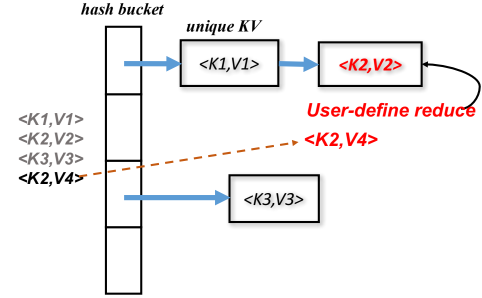


Fig. 6: Design of partial-reduction in Mimir.

Mimir introduces data partial-reduction as an optimization method for such cases, as shown in Figure 6. The optimization is exposed as an additional user callback function that the user can set, if desired. This callback function would then replace the `convert` and `reduce` phases. The semantics of the partial-reduction callback function are as follows. Mimir scans the KVs and hashes them to buckets based on the key. When it encounters a KV with a key that is already present in the bucket, the partial-reduction callback is called, which reduces these two KVs into a single KV. The existing KV in the hash bucket then is replaced with the reduced version. We note that the partial-reduction callback is called multiple times; in fact, it is called as many times as there are KVs with duplicate keys that need to be reduced.

2) KV Compression: KV compression is a common optimization used by many MapReduce frameworks, including MR-MPI. It is conceptually similar to the partial-reduction optimization. Like the partial-reduction optimization, this optimization is exposed as an additional user callback function that the user can set, if desired. The difference between KV compression and partial reduction, however, is that the KV compression callback function is called before the `aggregate` phase, instead of during the `reduce` phase.

The general working model of KV compression is similar to that of the partial-reduction optimization. When the `map` callback function inserts a KV, it is inserted into a hash bucket instead of the `aggregate` buffer. If a KV with an identical key is found, the KV compression callback function is called, which takes the two KVs and reduces them to a single KV. The existing KV in the hash bucket then is replaced with the reduced version.

The goal of the KV compression optimization is to reduce the size of the KVs before the `aggregate` phase. As a result, the data that is sent over the network in the `aggregate` phase is greatly reduced. Since KV compression is used during the `map` phase rather than the `reduce` phase, it can be applied to a broader range of jobs, including map-only jobs.

We note, however, that KV compression has some downsides. First, KV compression uses extra buffers to store the hash buckets. Thus, it reduces memory usage only if the compression ratio reaches a certain threshold. Second, it introduces extra computational overhead. Third, in Mimir when KV compression is enabled, the `aggregate` phase is delayed until all KVs are compressed to maximize the benefit

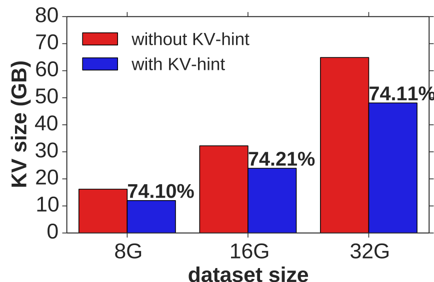


Fig. 7: KV size of *WordCount* with Wikipedia dataset.

of compression. This third shortcoming is an implementation issue and not a fundamental shortcoming of KV compression itself, and we hope to improve it in a future version of Mimir. While we implemented KV compression in Mimir for completeness and compatibility with MR-MPI, based on these shortcomings we caution users from trying to overexploit this functionality.

3) KV-hint: The key and value in a KV are conventionally represented as byte sequences of variable lengths, for generality. As a result, in Mimir we add an eight-byte header (two integers), containing the lengths of the key and value, before the actual data of the KV. For some datasets, however, these keys and values are fixed-length types; for example, in some graph processing applications, vertices and edges are always 64-bit and 128-bit integers, respectively. In this case, storing the lengths for every key and value is highly redundant and unnecessary. Mimir introduces an optimization called KV-hint that allows users to tell Mimir that the length of the key and value are constant for all keys. We implemented the KV-hint optimization in the KVC so that the KVCs used by different MapReduce functions can have their own setting of key and value lengths.

Mimir provides interfaces for the user to indicate whether the key or value has a fixed length throughout the entire job. For example, the key in the *WordCount* application is usually a string with variable length, but the value is always a 64-bit integer. In this case, the user can provide a hint to Mimir that the length of the value will always be 8 bytes. We reserve a special value of `-1` to indicate that the key or value is a string with a null-character termination. While the length of the string is variable in this case, it can be internally computed by using the `strlen` function and thus does not need to be explicitly stored. Figure 7 shows the memory usage of the *WordCount* application while processing the Wikipedia dataset [8]. The KV-hint optimization can save close to 26% memory for the KVs. As an unexpected side benefit, this optimization also reduces the amount of data that needs to be communicated during the *aggregate* phase, thus improving performance.

IV. Evaluation

In this section, we evaluate Mimir with respect to memory usage and performance and compare it with MR-MPI.

A. Platforms, Benchmarks, and Settings

Our experiments were performed on two different platforms: the XSEDE cluster Comet [2] and the IBM BG/Q supercomputer Mira [4]. Comet is an NSF Track2 system located at the San Diego Supercomputer Center. Each compute node has two Intel Xeon E5-2680v3 CPUs (12 cores each, 24 cores total) running at 2.5 GHz. Each node has 128 GB of memory and 320 GB of flash SSDs. The nodes are connected with Mellanox FDR InfiniBand, and the parallel file system is Lustre. Mira is an IBM BG/Q supercomputer located at Argonne National Laboratory. It has 786,432 compute nodes. Each node has 16 1.6 GHz IBM PowerPC A2 cores and 16 GB of DRAM. The nodes are connected with a 5D torus proprietary network, and the parallel file system is GPFS. Mira uses I/O forwarding nodes, with a compute-to-I/O ratio of 1:128; that is, each I/O forwarding node is shared by 128 compute nodes. We used MPICH 3.2 [6] for the experiments.

For our evaluation, we used three benchmarks: *WordCount* (WC), *octree clustering* (OC), and *breadth-first search* (BFS).

WC is a single-pass MapReduce application. It counts the number of occurrences of each unique word in a given input file. We tested WC with two datasets: (1) a uniform dataset of words (*Uniform*), which is a synthetic dataset whose words are randomly generated following a uniform distribution, and (2) the Wikipedia dataset (*Wikipedia*) from the PUMA dataset [8], which is highly heterogeneous in terms of type and length of words.

OC is an iterative MapReduce application with multiple MapReduce stages. As the application name suggests, OC is essentially a clustering algorithm for points in a three-dimensional space. We use the MapReduce algorithm described by Estrada et al. [11] for classifying points representing ligand metadata from protein-ligand docking simulations. The original application was written in MR-MPI. We ported it to Mimir for our experiments. The dataset is open source and described by Zhang et al. in [31]. In the dataset, the position of the points follows a normal distribution with a 0.5 standard deviation and a 1% density, meaning that the MapReduce library searches for and finds regions that have more than 1% of the total points.

BFS is an iterative map-only application. It is a graph traversal algorithm that generates a tree rooted at a source vertex. BFS is one of the three kernels of the Graph500 benchmark [20] and is a popular benchmark for evaluating supercomputer performance for data-intensive applications. We used the graph generator of the Graph500 benchmark to generate the BFS data. The graphs that are generated are *scale free* (i.e., the distribution of the edges follows a power law) with an average degree of 32 (i.e., the ratio of edges to vertices).

Of the optimizations presented in Section III-C, the KV-hint and KV compression optimizations were applied to all three benchmarks, while the partial-reduction optimization could be applied only to WC and OC. In this section, we use *hint*, *pr*, and *cps* when referring to the *KV-hint*, *partial-reduction*, and *KV compression* optimizations, respectively. While Mimir can

potentially set the page size to a small number in order to maximize page use, we set the size to 64 MB for all tests to ensure a fair comparison with MR-MPI, which uses 64 MB as the default page size. We also set the communication buffer to 64 MB to be consistent with the send buffer in MR-MPI.

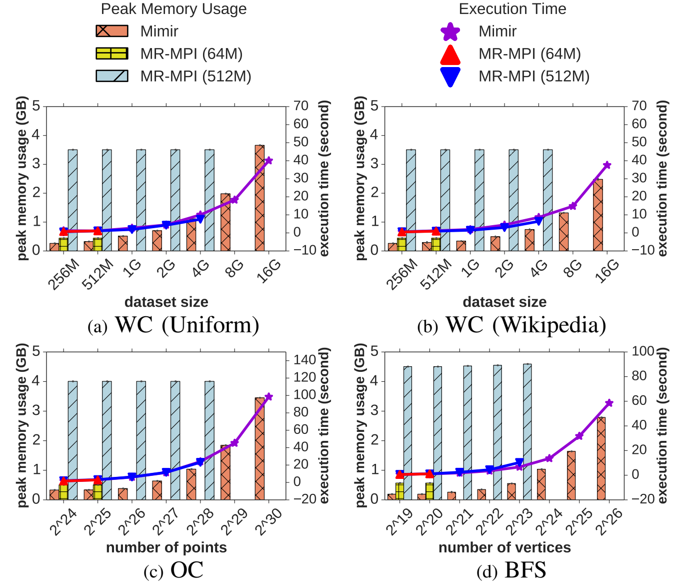
Our metrics of success in this evaluation are peak memory usage and execution time. Peak memory usage is the maximum memory usage at any point in time during the application execution. Execution time is the time from reading input data to getting the final results of a benchmark. The input data is stored in the parallel file system of our experimental platforms. When comparing Mimir with MR-MPI, times were measured for the two frameworks when the tests were performed in memory (i.e., no process spills data to the I/O subsystem). When performance metrics are missing in our results, the reason is that the associated test ran out of memory and spilled over to the I/O subsystem, thus causing substantial performance degradation (which can be measured in orders of magnitude of performance degradation).

B. Baseline Comparison with MR-MPI

In this section, we evaluate the core functionality and optimizations present in Mimir (Sections III-A and III-B), not including the optional optimizations from Section III-C. We consider this the "baseline" implementation of Mimir and compare it with MR-MPI in terms of peak memory usage and execution time (on a single node) and weak scalability (on multiple nodes). The page size of MR-MPI was set to 64 MB and 512 MB on Comet and to 64 MB and 128 MB on Mira. The default page size for MR-MPI is 64 MB; 512 MB and 128 MB are the maximum sizes possible for MR-MPI pages on Comet and Mira, so that MR-MPI can use all of the memory on these platforms, respectively.

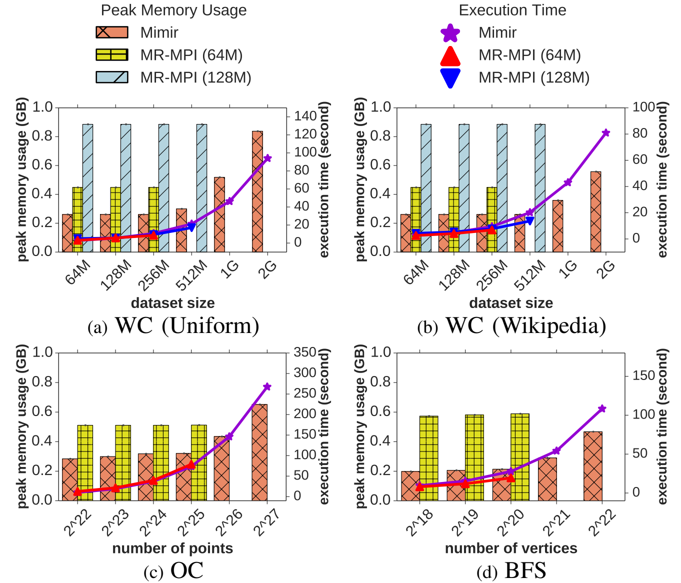
Figure 8 shows the memory usage and job execution times for all three benchmarks running on a single node of Comet. For WC with a uniform dataset, when both frameworks can run in memory, Mimir always uses less memory than MR-MPI does (e.g., at least 25% less memory compared with MR-MPI (64 MB)). For datasets larger than 512 MB, MR-MPI (64 MB) runs out of memory; and for datasets larger than 4 GB, MR-MPI (512 MB) runs out of memory. Mimir, on the other hand, supports in-memory computation for up to 16 GB datasets (i.e., 4-fold larger than the best case of MR-MPI). This improvement is due solely to Mimir's workflow, which uses memory more efficiently. Similar results are observed for WC with the Wikipedia dataset. For OC and BFS, Mimir still uses less memory (i.e., at least 34% and 64% less than MR-MPI (64 MB) does) for small datasets and allows execution of larger datasets (i.e., 4-fold larger for OC and 8-fold larger for BFS) compared with MR-MPI. As long as the dataset can be computed in memory, the execution times of the two frameworks are comparable. Once the dataset can no longer be computed in memory, we observe a substantial degradation of the performance for the impacted framework (data not shown in the figure).

Figure 9 shows the memory usage and job execution times of the three benchmarks running on a single node of Mira.



(a) WC (Uniform) (b) WC (Wikipedia) (c) OC (d) BFS

Fig. 8: Peak memory usage and execution times on one Comet node.

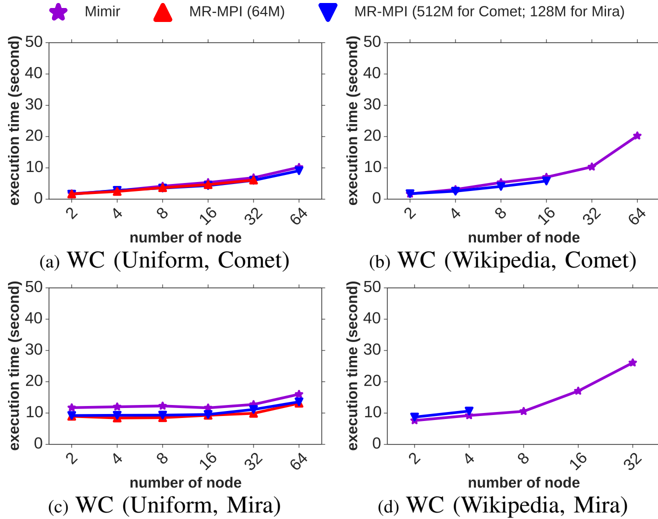


(a) WC (Uniform) (b) WC (Wikipedia) (c) OC (d) BFS

Fig. 9: Peak memory usage and execution times on one Mira node.

We see the same trend on Mira as on Comet in terms of more efficient use of memory (i.e., with a minimum gain of 40% across all tests), increased dataset sizes (i.e., 4-fold larger for all benchmarks), and similar performance for in-memory executions. Tests with 128 MB page sizes for MR-MPI were not performed for OC and BFS because MR-MPI runs out of memory.

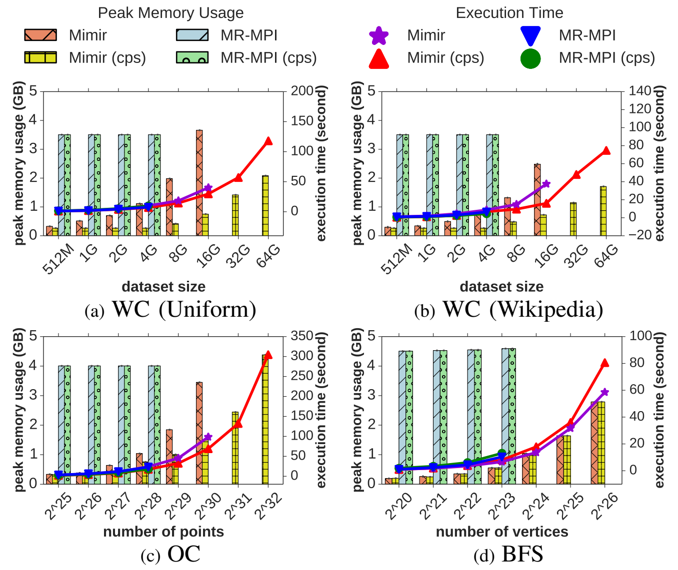
We also studied the weak scalability of the three benchmarks on Comet and Mira. We show in Figure 10 the results for the WC benchmarks. In Figures 10a and 10b, we keep the input data size per node to 512 MB because it is the largest dataset that MR-MPI (64 MB) configurations can run on the 24 processes of Comet. In Figures 10c and 10d, we keep the input data size per node to 256 MB because it is the largest



(a) WC (Uniform, Comet) (b) WC (Wikipedia, Comet)

(c) WC (Uniform, Mira) (d) WC (Wikipedia, Mira)

Fig. 10: Weak scalability of MR-MPI and Mimir.



(a) WC (Uniform) (b) WC (Wikipedia)

(c) OC (d) BFS

Fig. 11: Performance of KV compression on one Comet node.

dataset that MR-MPI (64 MB) configurations can run on the 16 processes of Mira. On Comet, Mimir can easily scale up to 64 nodes.[1] On the other hand, MR-MPI (64 MB) can scale up only to 32 nodes for WC (Uniform) and cannot scale up to even 2 nodes for the highly imbalanced dataset in WC (Wikipedia). When the page size increases from 64 MB to 512 MB, MR-MPI still scales up only to 16 nodes for WC (Wikipedia). The loss in scalability for MR-MPI is due to data imbalance: some processes have more intermediate data, thus exceeding the page size and spilling to the I/O subsystem. For a fair comparison between Mimir and MR-MPI across platforms, we also ran the scalability study on up to 64 nodes on Mira. We see similar trends in scalability on Mira as on Comet: Mimir exhibits good scalability on Mira, while MR-MPI with both 64 MB and 128 MB page sizes scales poorly for imbalanced datasets.

Scalability studies of OC and BFS on Comet and Mira (not shown in the paper) confirm the conclusions observed for WC.

C. Performance of KV Compression

In this section and in Section IV-D, we evaluate the optional optimizations that we presented in Section III-C. Of the three optimizations presented, only KV compression is available in MR-MPI. Thus, for fairness, we compared Mimir with MR-MPI only with this optimization enabled. The other two optimizations (i.e., partial reduction and KV-hints) are not enabled here. In Section IV-D we showcase the capabilities of all three optional optimizations in Mimir.

We compared the impact of the KV compression optimization on the memory usage of the three benchmarks when using a single node on Comet and Mira. We used the maximum page size from the previous comparisons for MR-MPI (i.e., on Comet we used 512 MB for all benchmarks; on Mira we used 128 MB for WC (Uniform) and WC (Wikipedia) and 64 MB for OC and BFS), because the increased page size allows MR-MPI to support larger datasets. For Mimir, we set the page size and buffers to 64 MB.

[1]We note that 64 is the maximum number of nodes available to XSEDE users on Comet.

Results for peak memory usage and execution time on a single Comet node are shown in Figure 11. The KV compression implementation of Mimir reduces the peak memory usage and allows processing more data in memory compared with the baseline Mimir as well as with MR-MPI (with and without KV compression) for WC (Uniform), WC (Wikipedia), and OC. The reason is that the buffers in Mimir are carefully managed: when KV compression reduces the size of the intermediate data, the empty buffers are freed to reclaim the memory. For BFS, Mimir has the same memory usage with and without compression because the compression reduces the size of data only during the graph traversal phase of the benchmark, while the peak memory usage occurs in the graph partitioning phase, which remains unaffected. Still, Mimir has a smaller, more efficient memory usage than does MR-MPI. For both WC (Uniform) and WC (Wikipedia) the figure shows datasets of up to 64 GB; we note, however, that with KV compression Mimir can support even larger datasets. KV compression in Mimir improves the performance of WC (Uniform), WC (Wikipedia), and OC but not that of BFS because of the additional computational cost in KV compression, as described in Section III-C2. With MR-MPI we do not observe any impact on peak memory usage because, despite the compression, the framework uses a fixed number of pages. In other words, the compression just reduces the shuffled data but does not impact the memory usage. Therefore, MR-MPI cannot support larger datasets as Mimir can.

The single-node results on Mira are shown in Figure 12. The peak memory usage and execution time patterns are similar to those on Comet with Mimir, while processing up to 16-fold larger datasets compared with MR-MPI when using KV compression.

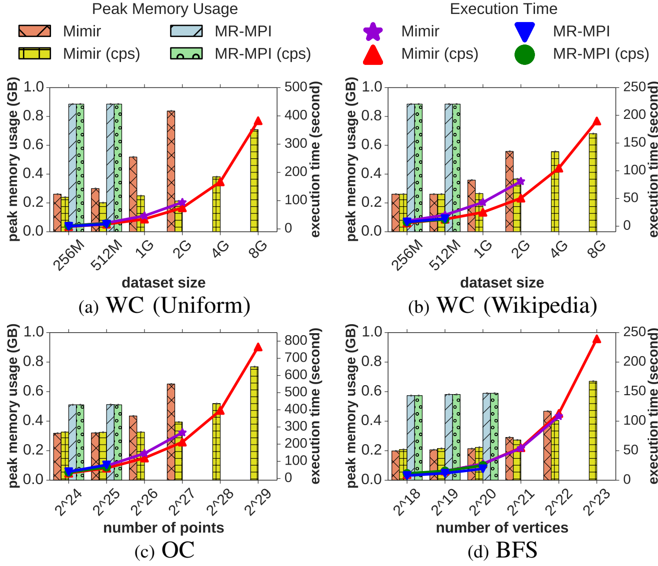


Fig. 12: Performance of KV compression on one Mira node.

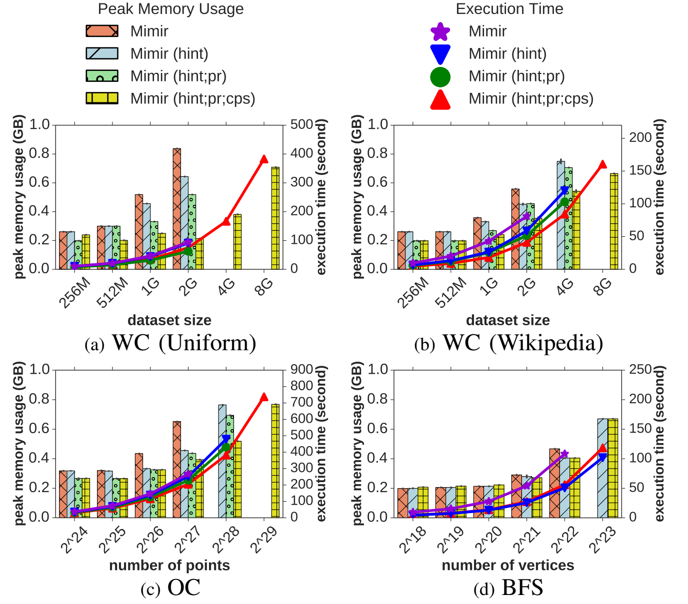


Fig. 13: Performance of different optimizations on one Mira node.

D. Impact of Optional Optimizations in Mimir

As mentioned earlier, MR-MPI does not provide the partial-reduction and KV-hint optimizations described in Section III-C). Thus we focus here on understanding the limits of Mimir alone, with respect to memory usage, execution time, and scalability. We measured the peak memory usage and execution time on a single node of Mira, as well as weak scalability on up to 1,024 nodes of this supercomputer. The page size and buffers used on Mimir were set to 64 MB.

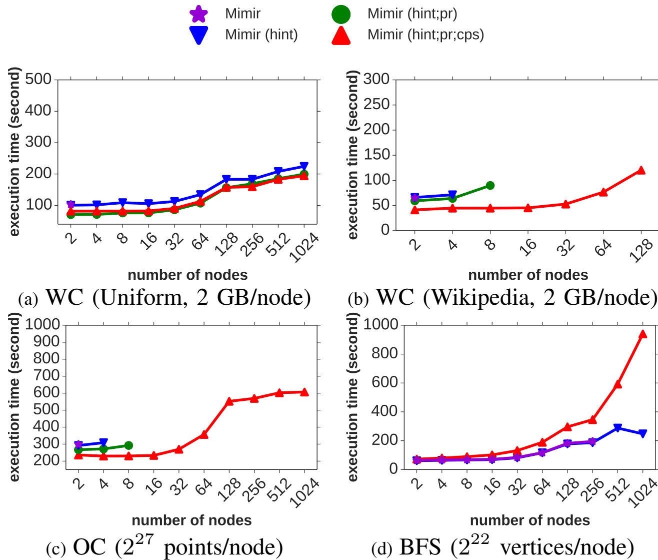
Single-node results on Mira are shown in Figure 13. We see that starting with the baseline implementation of Mimir and by adding the KV-hint, partial-reduction, and KV compression optimizations, one at a time, the peak memory usage reduces accordingly for the WC (Uniform), WC (Wikipedia), and the OC benchmarks. The BFS algorithm used by Mimir does not support the partial-reduction optimization. BFS has a reduction in memory usage with KV-hint but no improvement with KV compression, as outlined in Section IV-C. The three optimizations not only increase the amount of data that can be processed on a single node of Mira—4-fold larger for WC (Uniform), WC (Wikipedia), and OC and 2-fold larger for BFS compared with the baseline implementation—but also improve the performance of WC (Uniform), WC (Wikipedia), and OC. The KV-hint optimization also improves the performance of BFS.

We studied the weak scalability of Mimir on up to 1,024 nodes (i.e., 16,384 cores) of Mira. We used 2 GB/node for the two WC settings, 2^{27} points/node for OC, and 2^{22} vertices/node for BFS as the dataset size per node; these represent the maximum dataset sizes that the Mimir baseline implementation can process on each node. Because the maximum Wikipedia data that we can download from the PUMA dataset is small (≈400 GB), we can test the weak scalability of WC (Wikipedia) on up to 128 nodes only. As shown in Figure 14, some versions of Mimir cannot scale to 1,024 nodes because of the load imbalance of the data across MPI processes: load imbalances cause some processes to run out of memory. For example, the baseline implementation can scale up to only 2 nodes (32 MPI processes) for WC (Uniform), WC (Wikipedia), and OC before running out of memory. The baseline implementation of BFS scales up to 256 nodes (4,096 MPI processes). After applying the KV-hint optimization, WC (Uniform) and BFS scale up to 1,024 nodes; WC (Wikipedia) and OC scale up to only 4 nodes. The latter two benchmarks exhibit a much more severe load imbalance for the datasets that we are using in our tests. By adding the partial-reduction optimization, WC (Wikipedia) and OC scale up to 8 nodes. Only after applying the KV compression optimization does WC (Wikipedia) scale up to 128 nodes and OC up to 1,024 nodes. In general, the optimizations improve the overall execution times for WC (Uniform), WC (Wikipedia), and OC. A few exceptions exist: for example, when applied to WC (Uniform) and BFS, the KV compression optimization does not improve performance because of the extra compression overhead.

V. Related Work

Apart from MR-MPI, other implementations of MapReduce over MPI exist. *K*-MapReduce is a MapReduce framework developed and optimized for the K supercomputer [18]. Contrary to *K*-MapReduce, Mimir was designed keeping in mind a broader range of scientific computing platforms and applications.

Smart [25] is a MapReduce-like system for in situ data processing on supercomputing systems. *Smart* does not provide the complete MapReduce semantics and its programming interfaces, however, thus departing from the traditional MapReduce model to serve the needs of in situ data processing. While *Smart* certainly has a role to play in its target domain, it is



(a) WC (Uniform, 2 GB/node) (b) WC (Wikipedia, 2 GB/node)

(c) OC (2^{27} points/node) (d) BFS (2^{22} vertices/node)

Fig. 14: Weak scalability of different optimizations on Mira.

not a valid replacement for applications relying on the full semantics of MapReduce. Mimir, on the other hand, rigorously supports the MapReduce model while still enabling efficient in situ data analytics on supercomputing systems.

DataMPI [16] is a proposal to extend MPI to support MapReduce-like communication. The proposal is a work in progress, although it is unlikely to be integrated into the MPI standard mainly because it does not articulate the need to integrate such functionality into the MPI standard as opposed to implementing it as a high-level library above MPI. That is, the key question as to whether one can do MapReduce more portably or efficiently by integrating DataMPI into the MPI standard has not been answered by the researchers. In contrast to DataMPI, Mimir implements the MapReduce model as a lightweight portable framework on top of MPI.

Hoefler et al. [14] discuss using advanced MPI features to optimize MapReduce implementations. Mohamed et al. [19] discuss overlapping map and reduce functions in MapReduce over MPI. These optimizations focus on improving the performance of the *shuffle* communication. While such performance optimizations are important for any MapReduce framework, they are orthogonal to the memory efficiency improvements that Mimir targets.

Efforts targeting cloud-based MapReduce frameworks on supercomputing systems include adaptations of Hadoop [28] and Spark [26], [28], acceleration of communication by using RDMA [17], and tuning scalability [9] in cloud-based MapReduce frameworks. These projects, while working on the underlying implementation of Hadoop and Spark, provide unmodified Java/Python programming interfaces. Mimir's programming interface supports bindings for C and C++, which are often more suitable for scientific computing applications.

Other research has focused on implementing the MapReduce model on shared-memory systems. Phoenix [29], [23] targets thread-based parallel programming on shared-memory systems; Mars [13] is a MapReduce implementation on GPUs; and Mrphi [15] is a MapReduce implementation optimized for the Intel Xeon Phi. Different from these systems, Mimir works on large-scale distributed-memory systems.

VI. Conclusions

In this paper, we present Mimir, a memory-efficient and scalable MapReduce framework for supercomputing systems. Compared with other MPI-based MapReduce frameworks, such as MR-MPI, Mimir reduces memory usage significantly. The improved memory usage comes with better performance, ability to process larger datasets in memory (e.g., at least 16-fold larger for *WordCount*), and better scalability. Mimir's advanced optimizations improve performance and scalability on supercomputers such as Mira (an IBM BG/Q supercomputer). Overall, our results for three benchmarks, four datasets, and two different supercomputing systems show that Mimir significantly advances the state of the art with respect to efficient MapReduce frameworks for data-intensive applications. Mimir is an open-source software, and the source code can be accessed 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 was also supported by China Scholarship Council. Yutong Lu was supported by National Key R&D Project in China 2016YFB1000302. Part of the research in this paper used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility. XSEDE resources, supported by NSF grant ACI-1053575, were used to obtain some other performance data.

References

[1] Apahce Hadoop. http://hadoop.apache.org/.
[2] Comet Cluster. http://www.sdsc.edu/support/user_guides/comet.html.
[3] IBM BG/Q Architecture. https://www.alcf.anl.gov/files/IBM_BGQ_Architecture_0.pdf.
[4] Mira Supercomputer. https://www.alcf.anl.gov/mira.
[5] MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.
[6] MPICH Library. http://www.mpich.org.
[7] Turing: Memory Fragmentation Problem. http://www.idris.fr/eng/turing/turing-fragmentation_memoire-eng.html.
[8] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar. PUMA: Purdue MapReduce Benchmarks Suite. 2012.
[9] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan. Scaling Spark on HPC Systems. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 97–110, 2016.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[11] 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. *Computers in Biology and Medicine*, 42(7):758–771, 2012.
[12] Y. Guo, W. Bland, P. Balaji, and X. Zhou. Fault Tolerant Mapreduce-MPI for HPC Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[13] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 260–269, 2008.

[14] T. Hoefler, A. Lumsdaine, and J. Dongarra. Towards Efficient Mapreduce Using MPI. In *Proceedings of the European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 240–249. Springer, 2009.

[15] M. Lu, Y. Liang, H. P. Huynh, Z. Ong, B. He, and R. S. M. Goh. Mrphi: An Optimized MapReduce Framework on Intel Xeon Phi Coprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):3066–3078, 2015.

[16] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu. DataMPI: Extending MPI to Hadoop-like Big Data Computing. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[17] 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 *Proceedings of the 22nd Annual Symposium on High-Performance Interconnects*, pages 9–16, 2014.

[18] M. Matsuda, N. Maruyama, and S. Takizawa. K MapReduce: A Scalable Tool for Data-Processing and Search/Ensemble Applications on Large-Scale Supercomputers. In *Proceedings of the Cluster Computing Conference (CLUSTER)*, 2013.

[19] H. Mohamed and S. Marchand-Maillet. MRO-MPI: MapReduce Overlapping Using MPI and an Optimized Data Exchange Policy. *Parallel Computing*, 39(12):851–866, 2013.

[20] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the Graph 500. *Cray Users Group (CUG)*, 2010.

[21] S. J Plimpton and K. D. Devine. MapReduce in MPI for Large-Scale Graph Algorithms. *Parallel Computing*, 37(9):610–632, 2011.

[22] S.-J. Sul and A. Tovchigrechko. Parallelizing BLAST and SOM Algorithms with MapReduce-MPI Library. In *Proceedings of the 25th International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 481–489, 2011.

[23] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: Modular MapReduce for Shared-memory Systems. In *Proceedings of the 2nd International Workshop on MapReduce and Its Applications*, pages 9–16, 2011.

[24] M. Wasi ur Rahman, X. Lu, N. Sh. Islam, R. Rajachandrasekar, and D. K. Panda. High-Performance Design of YARN MapReduce on Modern HPC Clusters with Lustre and RDMA. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.

[25] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang. Smart: A MapReduce-like Framework for in-situ Scientific Analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[26] Y. Wang, R. Goldstone, W. Yu, and T. Wang. Characterization and Optimization of Memory-Resident MapReduce on HPC Systems. In *Proceedings of the 28th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 799–808, 2014.

[27] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[28] X. Yang, N. Liu, B. Feng, X.-H. Sun, and S. Zhou. PortHadoop: Support Direct HPC Data Processing in Hadoop. In *Proceedings of the International Conference on Big Data (Big Data)*, pages 223–232, 2015.

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

[30] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[31] 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 16th International Conference on Computational Science and Engineering (CSE)*, pages 117–124, 2013.