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

Authors: Tao Gao, Yanfei Guo, Boyu Zhang, Pietro Cicotti, Yutong Lu, Pavan Balaji, and Michela Taufera

2017 IEEE International Parallel and Distributed Processing Symposium

Presented By: Sumaira Erum Zaib (17M38242)

Outline

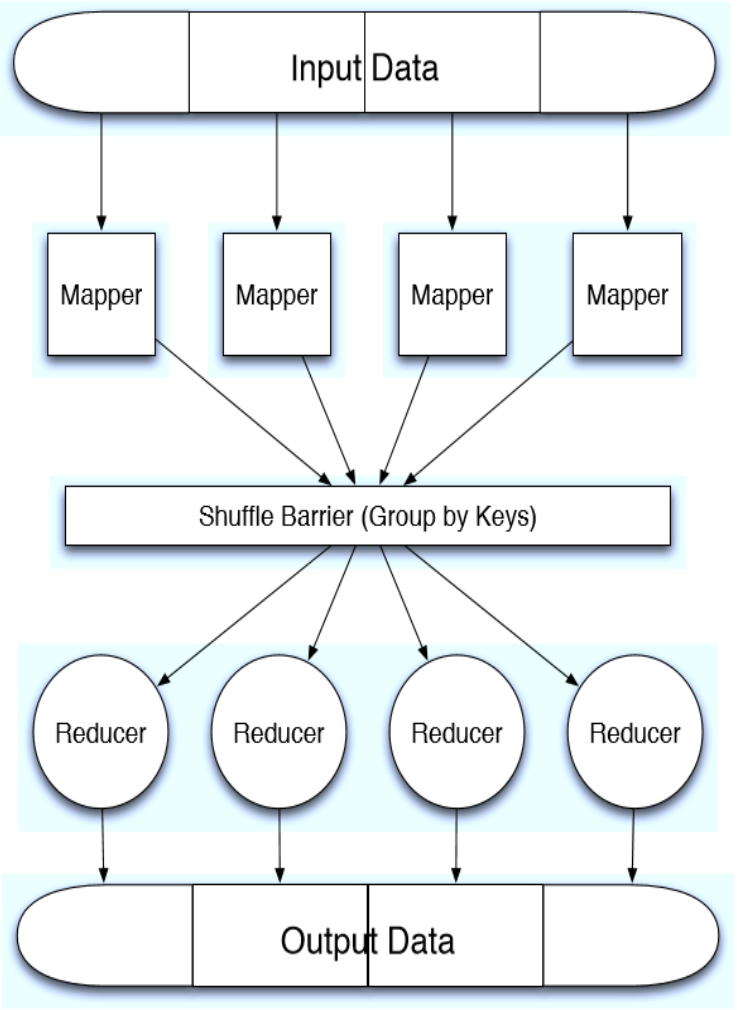
- MapReduce
- MR-MPI
- MR-MPI Memory Management
- Mimir
- Evaluation
- Conclusion

MapReduce

- MapReduce is a framework for processing parallelizable problems across large datasets using a large number of computers (nodes), collectively referred to as a cluster or a grid.
- Three phases: Map, Shuffle, Reduce
- A global barrier between each phase ensures correctness
- The user implements the map and reduce callback functions, while the MapReduce runtime handles the parallel job execution, communication, and data movement.

MapReduce: Map, Shuffle, Reduce

- Map():
  - processes the input data using a user-defined map callback function and generates intermediate key, value (KV) pairs.
- Shuffle():
  - performs an all-to-all communication that distributes the intermediate KV pairs across all processes. KV pairs with the same key are also merged and stored in <key, <value1, value2...>> (KMV) lists.
- Reduce():
  - processes the KMV lists with a user-defined reduce callback function and generates the final output.



MapReduce - Message Passing Interface (MR-MPI)

- Message Passing Interface (MPI)
  - standardized and portable message-passing standard
  - designed by a group of researchers from academia and industry
  - functions on a wide variety of parallel computing architectures.
  - The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran.

- MR-MPI is an implementation of MapReduce on top of MPI

MapReduce - Message Passing Interface (MR-MPI)

- Supports the logical map-shuffle-reduce workflow in four phases:
  - Map – same as before
  - Aggregate – handles all to all movement of data between processes. Here MR-MPI also calculates the data and buffer sizes and exchanges intermediate KV pairs.
  - Convert – After the exchange, convert phase merges all received KV pairs based on their keys
  - Reduce – same as before
- Map, Reduce – need to be implemented by user
  Aggregate, Convert – need to be explicitly invoked by user

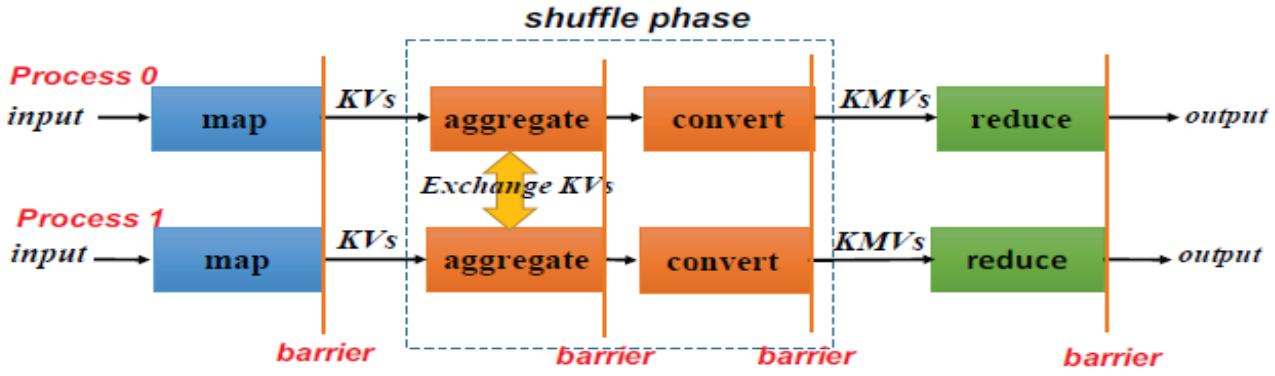


Fig. 2: The *map*, *shuffle*, and *reduce* phases in MR-MPI.

MR-MPI Memory Management

- MR-MPI uses a model based on fixed size buffer structure called page to store intermediate data (page is simply a large memory buffer and is not related to operating system pages).
- These pages are static memory buffers that are allocated at the start of each MapReduce phase and are used throughout the phase
- If the dataset can fit in these pages, the data processing is in memory. Otherwise, MR-MPI spills over the data into the I/O subsystem.
- While this model is functionally correct, it leads to tremendous loss in performance

MR-MPI Memory Management

- MR-MPI supports three out-of-core writing settings:
  - always write intermediate data to disk
  - write intermediate data to disk only when the data is larger than a single page
  - report an error and terminate execution if the intermediate data is larger than a single page size.
- 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.

MR-MPI Memory Management

- Similar to MapReduce framework, MR-MPI uses a global barrier to synchronize at the end of each phase
- Due to this barrier intermediate data is held 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
- For iterative MapReduce jobs, where the same dataset is repeatedly processed, buffers for intermediate data need to be repeatedly allocated and freed

MR-MPI Memory Management

- Frequent allocation and deallocation of memory buffers with different sizes can lead to memory fragmentation.
- Supercomputing systems such as IBM BG/Q use light weight kernels with simple memory manager that does not handle such memory fragmentation.
- To avoid memory fragmentation, MR-MPI uses pages. Usually larger pages are needed to use the system memory more effectively.
- For each MapReduce phase, MR-MPI tries to allocate all the pages it needs at once

MR-MPI Memory Management

- This coarse-grained memory allocation leads to another efficiency problem: not all allocated pages are fully utilized
- For some MapReduce jobs, the size of intermediate data decreases as it passes through different phases.
  - 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 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 Memory Management

- MR-MPI suffers from redundant memory buffers and unnecessary memory copies
  - KVs are partitioned 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.
  - MR-MPI allocates two pages for the receive buffer to prevent buffer overflow
  - The aggregate phase copies the received KVs to the input buffer of the succeeding convert phase.

MR-MPI Memory Management

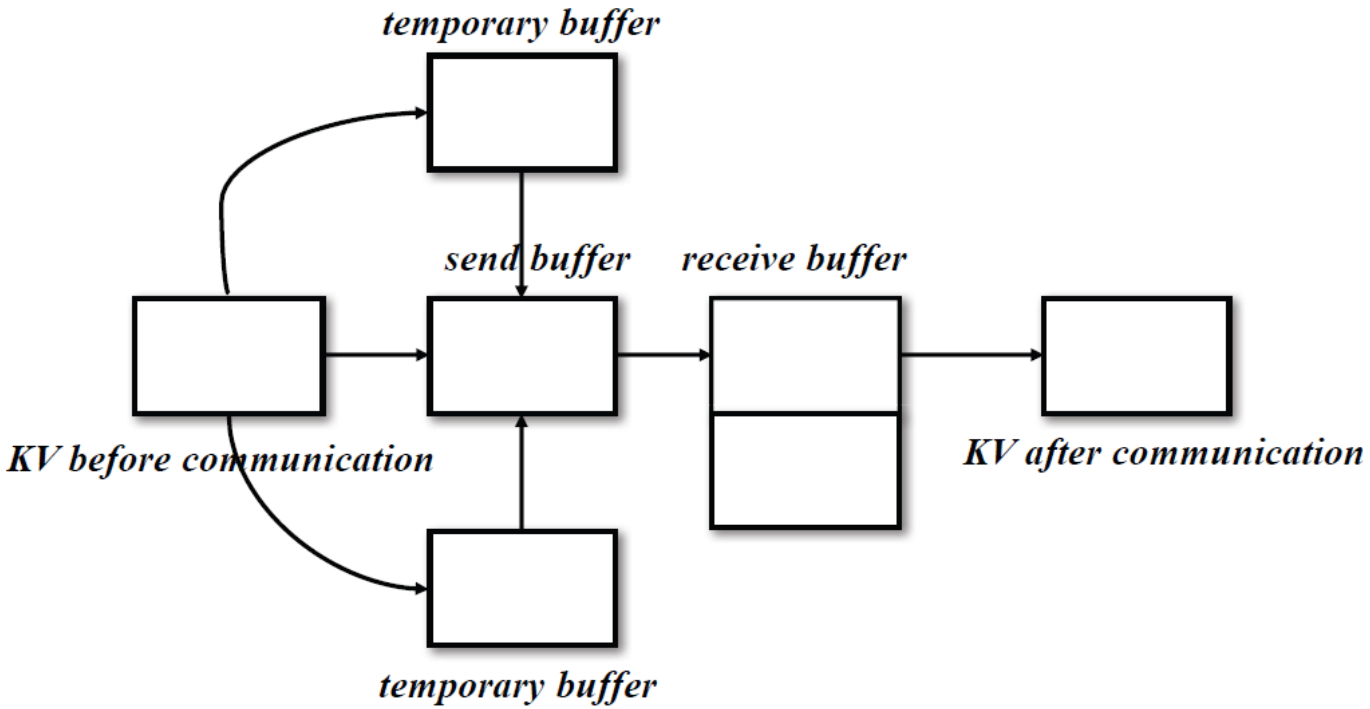
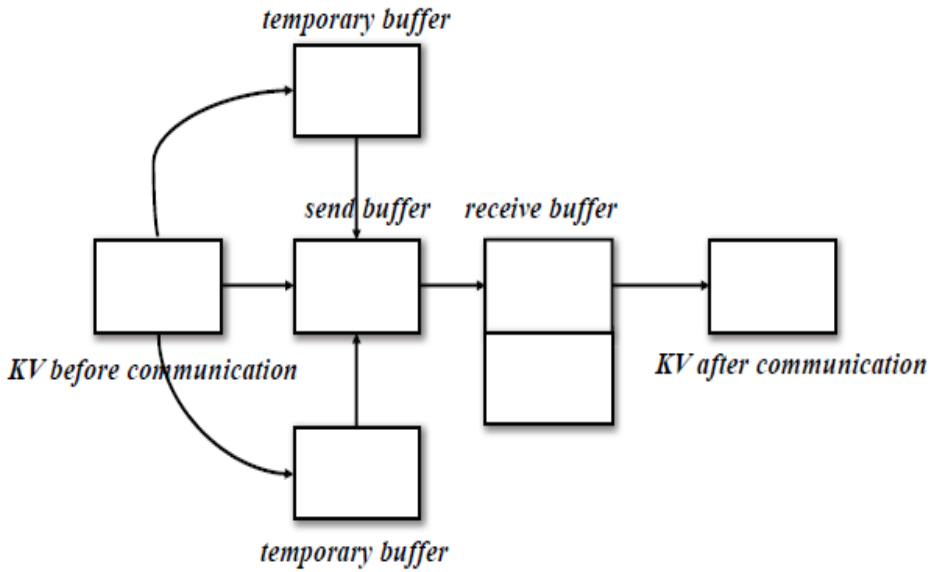


Fig. 3: Memory usage in *aggregate* phase.

MR-MPI Memory Management

- 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.



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 are "core" optimizations
  - The third class is "optional" optimizations

MIMIR

- 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
  - 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.

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: the aggregate and convert phases are implicit; that is, the user does not explicitly start these phases.
- This design offers two advantages:
  - 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.

Mimir Workflow Phases – Core Optimizations

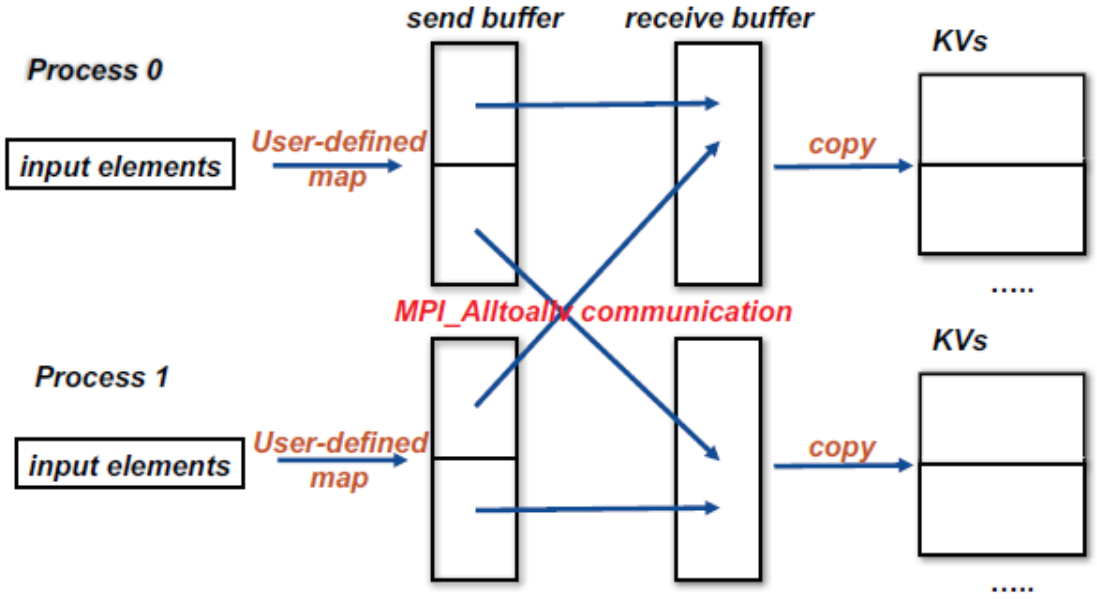


Fig. 4: Workflow of `map` and `aggregate` phases in Mimir.

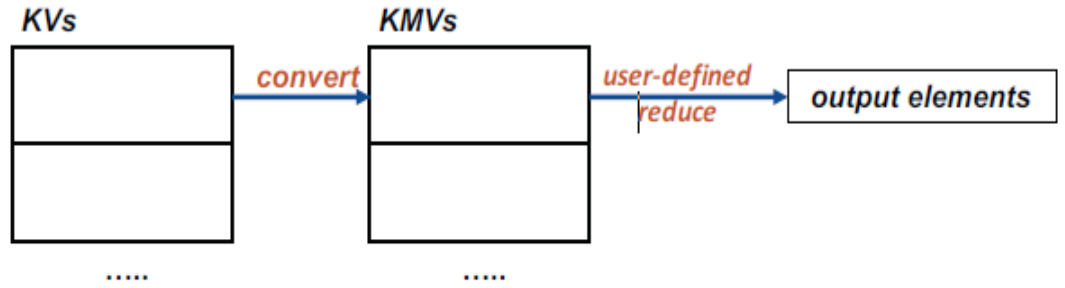


Fig. 5: Workflow of `convert` and `reduce` phases in Mimir.

Mimir Workflow Phases – Core Optimizations

- Map Phase:
  - 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.
  - If a partition in the send buffer is full the map phase is temporarily suspended and switched to the aggregate phase.

Mimir Workflow Phases – Core Optimizations

- 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 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.
  - Therefore, the map and aggregate phases are interleaved, allowing them to process large volumes of input data without correspondingly increasing the memory usage.

Mimir Workflow Phases – Core Optimizations

- Convert Phase:
  - The convert phase converts these KVs into KMVs and stores them in a KMVC.
  - A two-pass algorithm is used 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.
- Reduce Phase:
  - In this final phase, the user-defined reduce callback function are called on the KMVs.
  - We note that unlike the map and aggregate phases, the convert and reduce phases cannot be interleaved.

Mimir Memory Management – Core Optimizations

- Mimir uses two types of memory buffers:
  - data buffers for storing intermediate KVs and KMVs
  - communication buffers
- Unlike MR-MPI, Mimir allows data buffers to be dynamically allocated as the sizes of KVs and KMVs grow. KVCs and KMVCs were created 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.
- 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.

Mimir Workflow Hints – Optional Optimization

- These optimizations are not automatic and need to be explicitly requested for by the application because 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.
- Categories:
  - Advanced Functionality: the user application can implement additional callbacks that give the user more fine grained control of the data processing and movement.
    - Partial Reduction
    - KV Compression
  - Hints: giving a hint to the Mimir runtime about certain properties of the dataset being processed. There is no change to either the dataset or the computation on the dataset by the application
    - KV Hint

Partial Reduction

- The basic Mimir workflow performs the convert and reduce phases in a non-interleaved 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.
- 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:
  - 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.
  - 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.

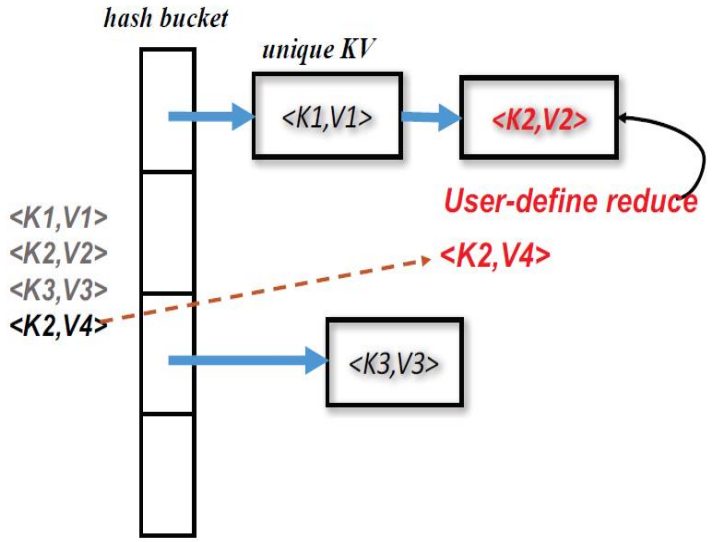


Fig. 6: Design of partial-reduction in Mimir.

KV Compression

- KV compression is conceptually similar to the partial-reduction optimization.
- The difference between KV compression and partial reduction is that the KV compression callback function is called before the aggregate phase, instead of during the reduce phase.
- 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.
- 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 of compression. This third shortcoming is an implementation issue and not a fundamental shortcoming of KV compression itself

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 an eight-byte header was added, 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.
- The KV-hint optimization is implemented 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.
- 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.

Evaluation - Platforms

- XSEDE cluster Comet
  - 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
  - 320 GB of flash SSDs.
  - The nodes are connected with Mellanox FDR InfiniBand,
  - the parallel file system is Lustre.
- IBM BG/Q supercomputer Mira
  - 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
  - the parallel file system is GPFS. (The General Parallel File System (GPFS) is a high-performance clustered file system developed by IBM. It can be deployed in shared-disk or shared-nothing distributed parallel modes. It is used by many of the world's largest commercial companies, as well as some of the supercomputers)
  - 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.
  - MPICH 3.2 (an MPI implementation) was used for the experiments.

Evaluation - Benchmarks

- WordCount (WC):
  - WC is a single-pass MapReduce application.
  - It counts the number of occurrences of each unique word in a given input file.
  - Two datasets:
    - a uniform dataset of words (Uniform), which is a synthetic dataset whose words are randomly generated following a uniform distribution,
    - the Wikipedia dataset (Wikipedia) from the PUMA dataset, which is highly heterogeneous in terms of type and length of words
- Octree Clustering (OC):
  - OC is an iterative MapReduce application with multiple MapReduce stages.
  - It is a clustering algorithm for points in a three-dimensional space.
- Breadth-First Search (BFS):
  - BFS is an iterative map-only application.
  - It is a graph traversal algorithm that generates a tree rooted at a source vertex

Evaluation - Settings

- 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
- Page size was set to 64 MB for all tests to ensure a fair comparison with MR-MPI, which uses 64 MB as the default page size. Also the communication buffer size was set to 64 MB to be consistent with the send buffer in MR-MPI.
- Metrics of success
  - 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 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).

Evaluation – Baseline Comparison with MR-MPI

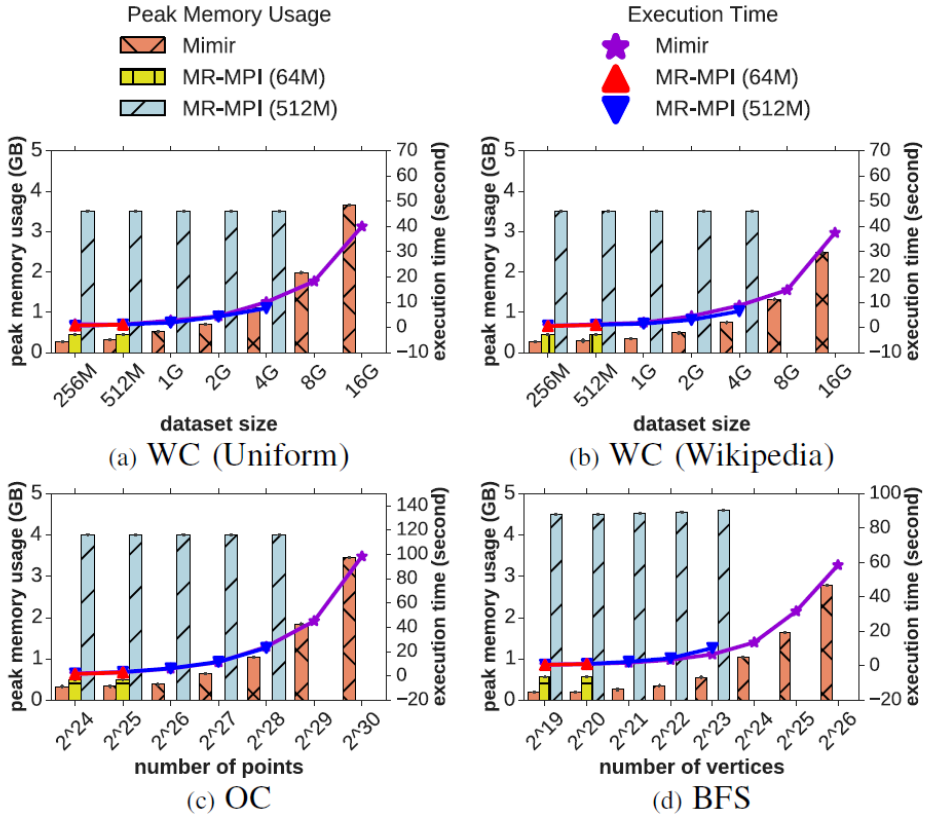


Fig. 8: Peak memory usage and execution times on one Comet node.

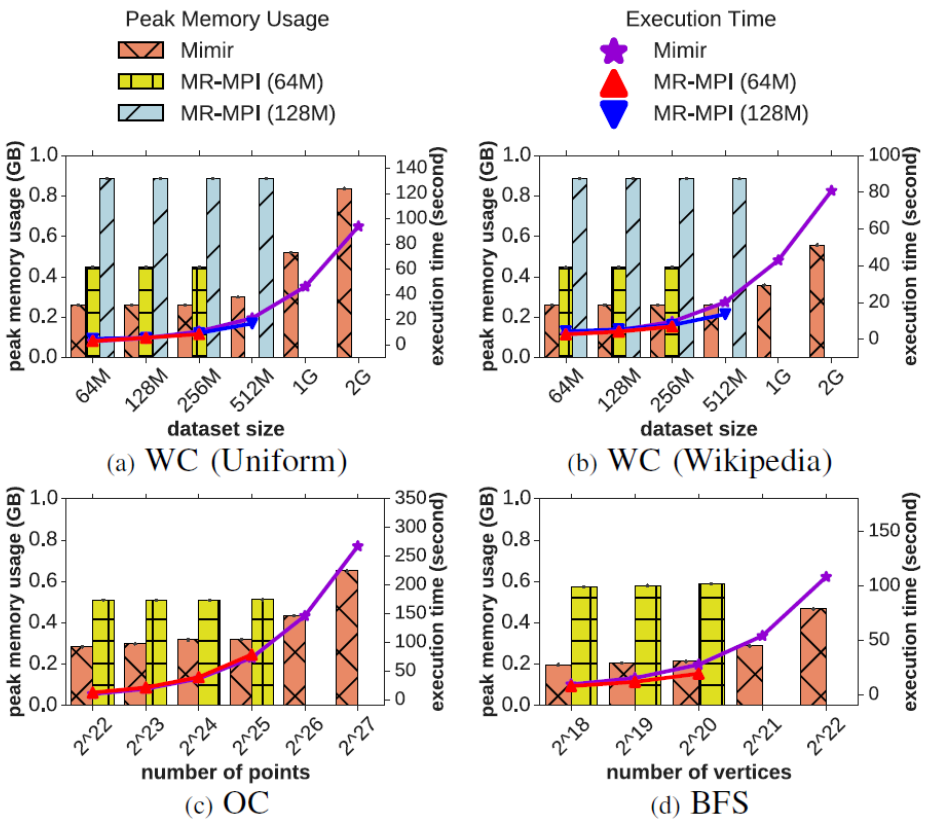
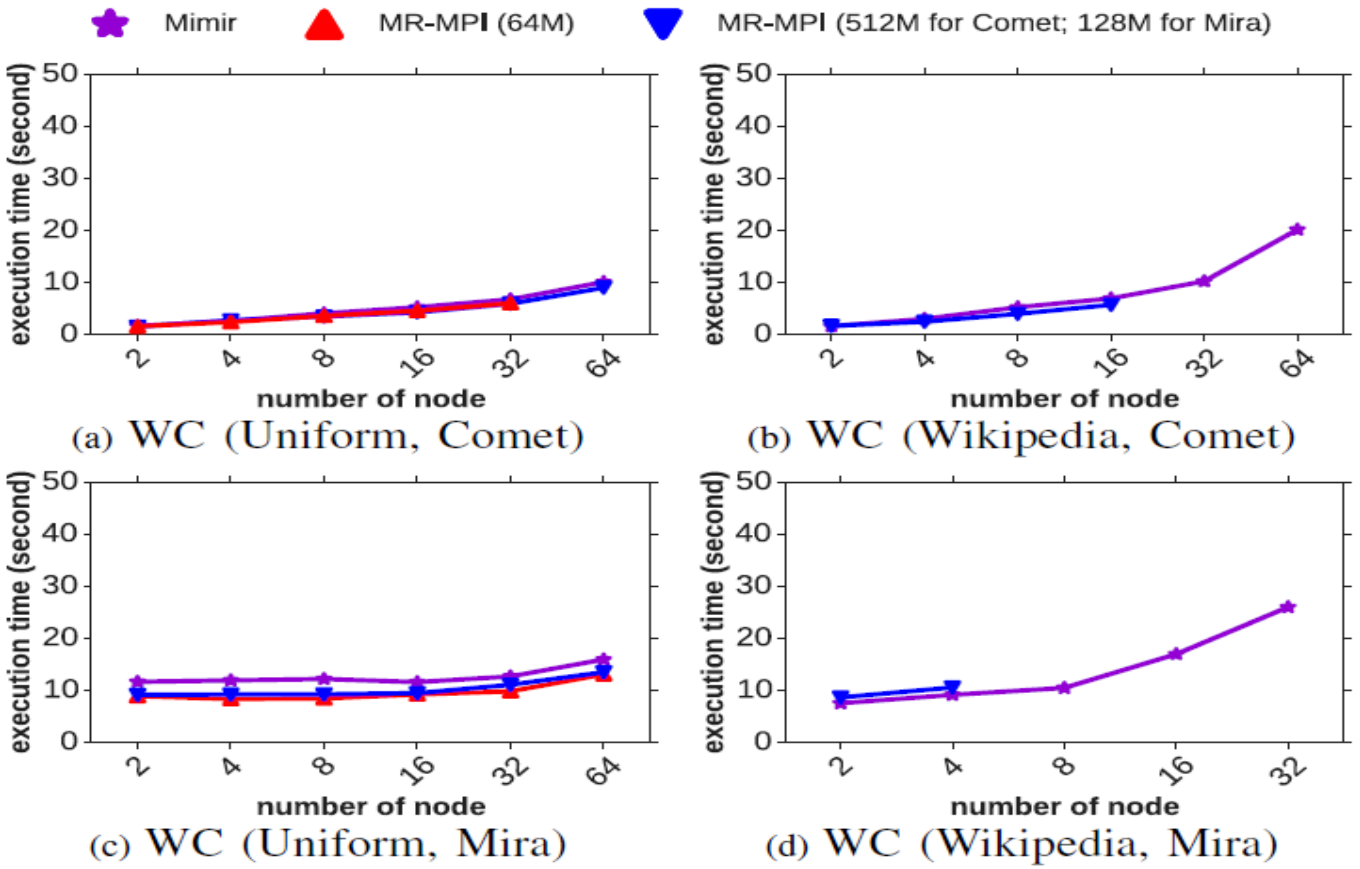


Fig. 9: Peak memory usage and execution times on one Mira node.

Evaluation – Baseline Comparison with MR-MPI



(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.

Evaluation – Performance of KV Compression

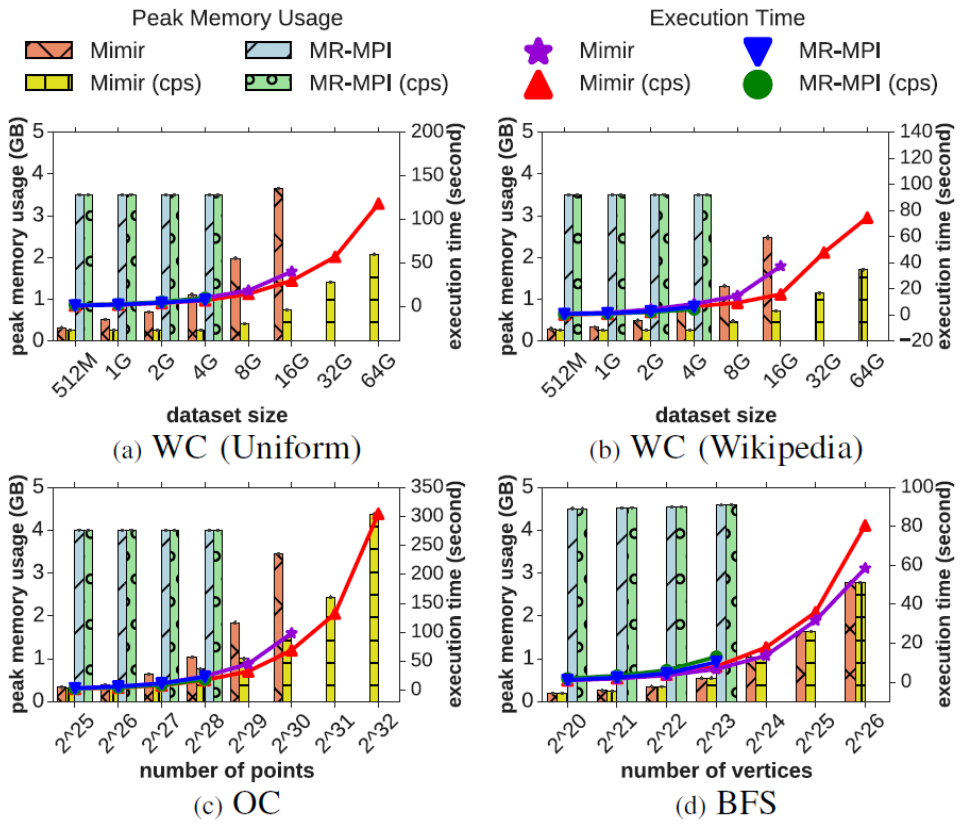


Fig. 11: Performance of KV compression on one Comet node.

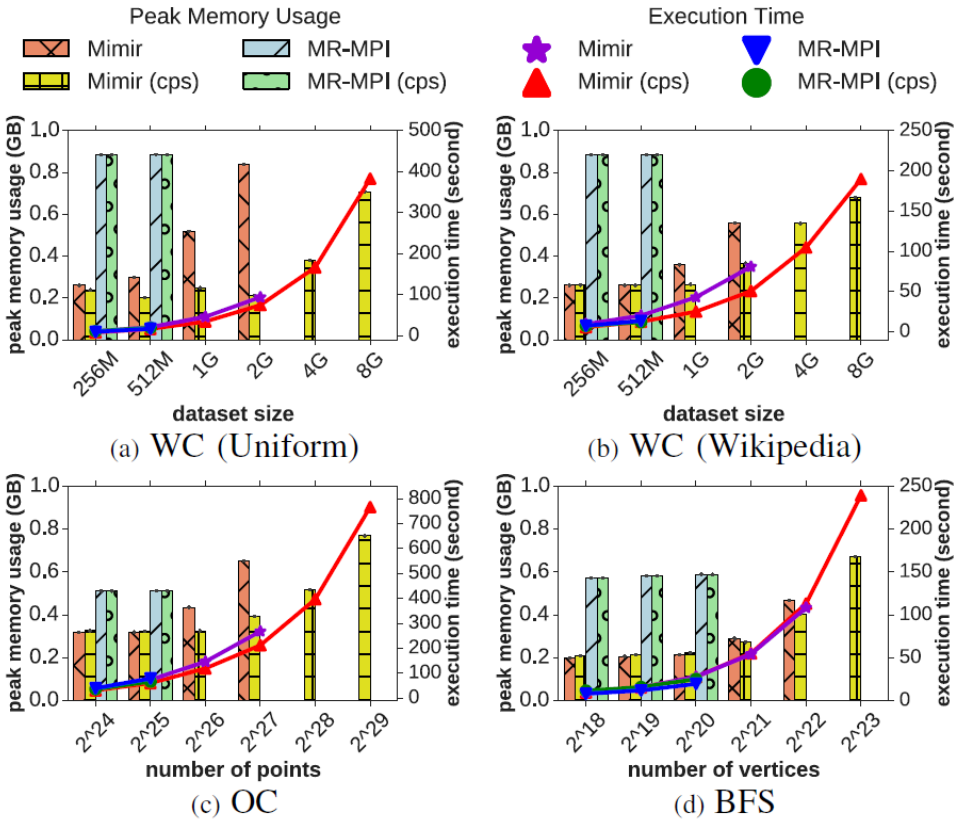
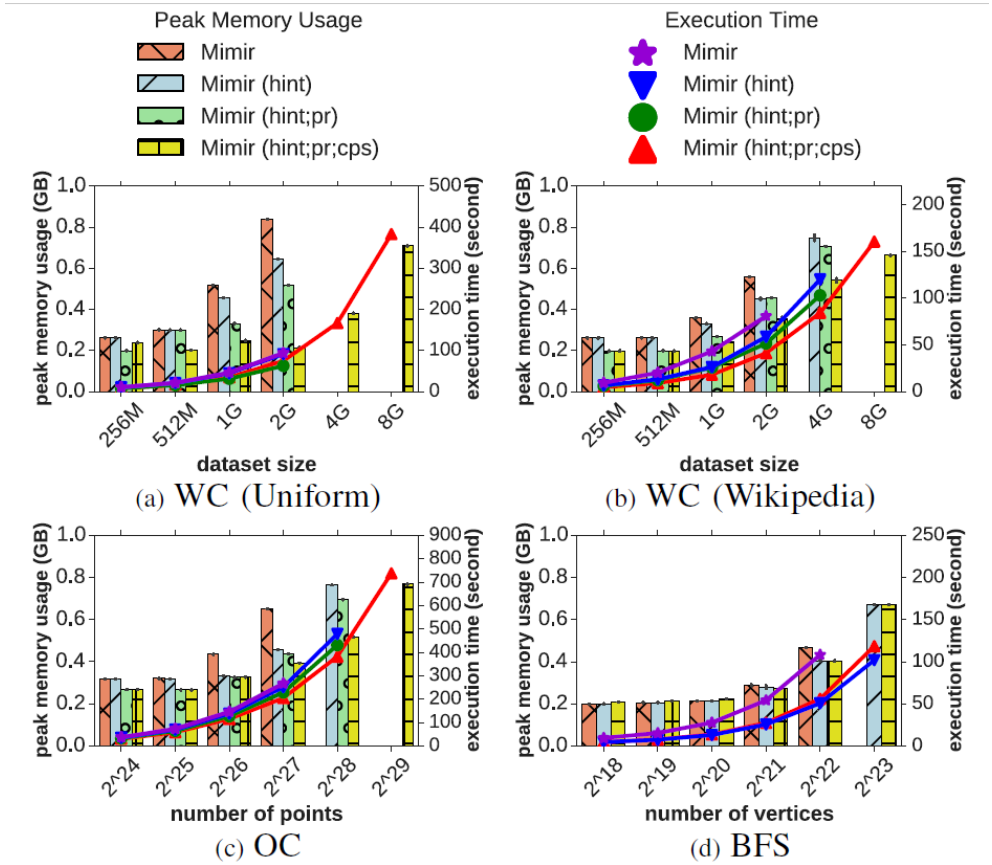


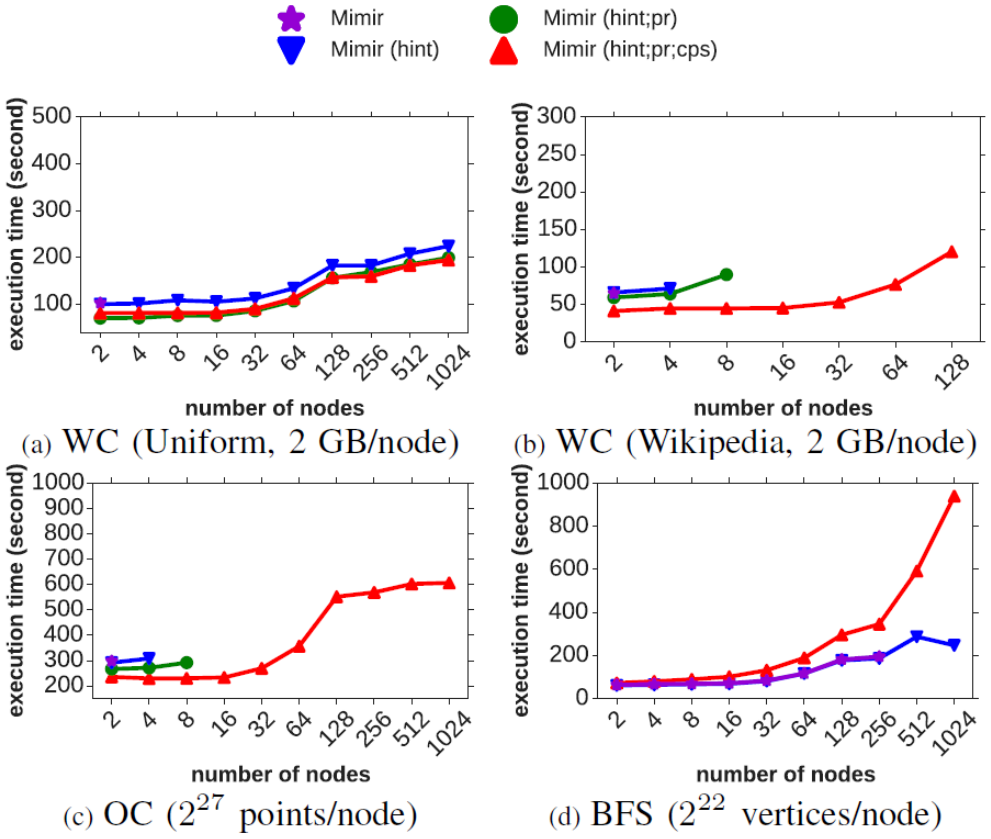
Fig. 12: Performance of KV compression on one Mira node.

Evaluation – Impact of Optional Optimizations on Mimir



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

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



(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.

Conclusion

- Compared with other MPI-based MapReduce frameworks, such as MR-MPI, Mimir
  - Reduces memory usage significantly
  - Has better performance
  - Can process larger datasets in memory (e.g., at least 16-fold larger for WordCount),
  - Has better scalability.
- Mimir's advanced optimizations improve performance and scalability on supercomputers such as Mira (an IBM BG/Q supercomputer). Overall, 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.

References

- Gao, Tao & Guo, Yanfei & Zhang, Boyu & Cicotti, Pietro & Lu, Yutong & Balaji, Pavan & Taufer, Michela. (2017). Mimir: Memory-Efficient and Scalable MapReduce for Large Supercomputing Systems. 1098-1108. 10.1109/IPDPS.2017.31.
- Verma, A., Cho, B., Zea, N. et al. Cluster Comput (2013) 16: 191. https://doi.org/10.1007/s10586-011-0182-7
- https://en.wikipedia.org/wiki/MapReduce