PapyrusKV: A High-Performance Parallel Key-Value Store for Distributed NVM Architectures

Jungwon Kim
Oak Ridge National Laboratory
kimj@ornl.gov

Seyong Lee
Oak Ridge National Laboratory
lees2@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
vetter@computer.org

ABSTRACT

This paper introduces PapyrusKV, a parallel embedded key-value store (KVS) for distributed high-performance computing (HPC) architectures that offer potentially massive pools of nonvolatile memory (NVM). PapyrusKV stores keys with their values in arbitrary byte arrays across multiple NVMs in a distributed system. PapyrusKV provides standard KVS operations such as put, get, and delete. More importantly, PapyrusKV provides advanced features for HPC such as dynamic consistency control, zero-copy workflow, and asynchronous checkpoint/restart. Beyond filesystems, PapyrusKV provides HPC programmers with a high-level interface to exploit distributed NVM in the system, and it transparently organizes data to achieve high performance. Also, it allows HPC applications to specialize PapyrusKV to meet their specific requirements. We empirically evaluate PapyrusKV on three HPC systems with real NVM devices: OLCF's Summitdev, TACC's Stampede, and NERSC's Cori. Our results show that PapyrusKV can offer high performance, scalability, and portability across these various distributed NVM architectures.

CCS CONCEPTS

• **Information systems** → **Key-value stores**; • **Hardware** → *Non-volatile memory*; • **Software and its engineering** → *Distributed programming languages*;

ACM Reference format:
Jungwon Kim, Seyong Lee, and Jeffrey S. Vetter. 2017. PapyrusKV: A High-Performance Parallel Key-Value Store for Distributed NVM Architectures. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 14 pages.
DOI: 10.1145/3126908.3126943

1 INTRODUCTION

Key-value stores (KVSs) have become a fundamental component of cloud computing, internet-scale databases, and content management [1–3, 5, 6, 10, 12, 14, 15, 17]. Their flexibility, reliability, and ease of use make them an important tool for semi-structured data in web and deep learning tasks. Recently, high-performance computing (HPC) has started to embrace this model for various scenarios [12, 26], such as coupling applications or storing intermediate results; however, existing KVS solutions are not optimized for HPC architectures or scientific applications. The use of KVSs in HPC brings challenges. First, KVSs must be scalable to, perhaps, millions of threads. Second, KVSs must interoperate with existing programming models for parallel applications. Third, KVSs must provide mechanisms for consistency and replication of data, but these mechanisms must be balanced against the needs for scalability and performance.

Meanwhile, future HPC architectures (e.g., the next-generation DOE supercomputers) will have massive pools of nonvolatile memory (NVM) that will dramatically increase node memory capacity [15, 18]. This NVM is emerging to fill an important role in HPC systems [25]: to increase node memory capacity while maintaining cost and energy efficiency and to provide buffers for parallel I/O where performance is not increasing at a rate that maintains a reasonable system balance. In this regard, NVM can offer very high capacity in-NVM KVSs in future HPC systems, providing a partial solution to this serious I/O imbalance.

1.1 Contributions

To address these trends, this paper introduces *PapyrusKV*, a parallel KVS for distributed HPC architectures that offer potentially massive pools of NVM. Like existing distributed KVSs, such as Bigtable [10], Cassandra [17], and HBase [1], PapyrusKV basically follows the log-structured merge-tree (LSM-tree) [20] that stores data in a hierarchical structure. More importantly, PapyrusKV presents several improvements over existing KVSs to provide advanced features for HPC. The key contributions of this paper are the following:

(1) We introduce PapyrusKV, a novel embedded KVS implemented specifically for HPC architectures and applications to provide scalability, replication, consistency, and high performance, and so that they can be customized by the application.
(2) PapyrusKV delivers high write and read performance by exploiting LSM-trees oriented for high write performance and NVMs that provide high-speed random access.
(3) PapyrusKV provides configurable consistency technique controlled by the application during the program execution dynamically to meet application-specific requirements and/or needs.
(4) PapyrusKV supports fault tolerance and streamlined workflow by leveraging NVM's persistence property.
(5) We empirically evaluate PapyrusKV on three different HPC systems (OLCF's Summitdev, TACC's Stampede, and NERSC's Cori), which are equipped with different NVM architectures, using microbenchmarks and a real HPC application (Meraculous [13]) to demonstrate its portability, scalability, and performance.

Table 1: The PapyrusKV API.

API Function	Description	Collective
(a) Environment		
papyruskv_init(int argc, char*** argv, const char* repository)*	Initialize execution environment using *repository* path	×
papyruskv_finalize()	Terminate execution environment	×
(b) Basic		
papyruskv_open(const char name, int flags, papyruskv_option_t* opt, papyruskv_db_t* db)*	Open or create *db* with *name*	×
papyruskv_close(papyruskv_db_t db)	Close *db*	×
papyruskv_put(papyruskv_db_t db, const char key, size_t keylen, const char* value, size_t valuelen)*	Insert or update a *key-value* pair to *db*	
papyruskv_get(papyruskv_db_t db, const char key, size_t keylen, char** value, size_t* valuelen)*	Retrieve *value* for a given *key* from *db*. If *value* is not allocated in memory, PapyrusKV allocates a new heap region from the PapyrusKV memory pool. Otherwise, data is copied to *value* directly	
papyruskv_delete(papyruskv_db_t db, const char key, size_t keylen)*	Delete a key-value pair for a given *key* from *db*	
papyruskv_free(papyruskv_db_t db, char val)*	Release a heap memory region allocated from the PapyrusKV memory pool	
(c) Consistency		
papyruskv_signal_notify(int signum, int ranks, int count)*	Send signals to *ranks*	
papyruskv_signal_wait(int signum, int ranks, int count)*	Wait for signals from *ranks*	
papyruskv_fence(papyruskv_db_t db)	Migrate the remote MemTable and immutable MemTables to the owner ranks immediately	
papyruskv_barrier(papyruskv_db_t db, int level)	Collective memory fence with a flushing *level* (`PAPYRUSKV_MEMTABLE` or `PAPYRUSKV_SSTABLE`). With `PAPYRUSKV_SSTABLE` level, the whole *db* data are flushed to SSTables.	×
papyruskv_consistency(papyruskv_db_t db, int mode)	Set memory consistency mode on *db* to *mode* (`PAPYRUSKV_SEQUENTIAL` or `PAPYRUSKV_RELAXED`)	×
papyruskv_protect(papyruskv_db_t db, int prot)	Set protection attribute on *db* to *prot* (`PAPYRUSKV_RDWR`, `PAPYRUSKV_WRONLY`, or `PAPYRUSKV_RDONLY`)	×
(d) Persistence		
papyruskv_checkpoint(papyruskv_db_t db, const char path, papyruskv_event_t* event)*	Generate a snapshot of *db* into *path*. It runs asynchronously if *event* is not NULL	×
papyruskv_restart(const char path, const char* name, int flags, papyruskv_option_t* opt, papyruskv_db_t* db, papyruskv_event_t* event)*	Revert *db* with *name* from a snapshot stored in *path*. It runs asynchronously if *event* is not NULL	×
papyruskv_destroy(papyruskv_db_t db, papyruskv_event_t event)*	Remove *db* and all its data from NVM. It runs asynchronously if *event* is not NULL	×
papyruskv_wait(papyruskv_db_t db, papyruskv_event_t event)	Wait for *event* to complete	×

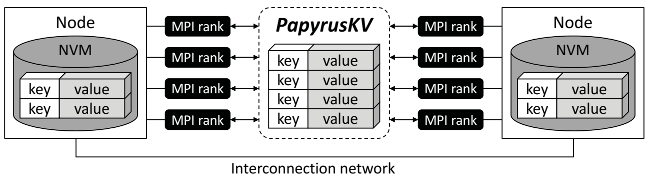

Figure 1: PapyrusKV overview.

2 PAPYRUSKV ARCHITECTURE

2.1 Design Goals

In designing PapyrusKV, we prioritized several key goals to drive our development of its features and capabilities.

Performance. One critical goal of PapyrusKV is high performance. Specifically, PapyrusKV must provide performance better than or equal to alternatives for providing large shared storage for an HPC application, *where overall application data requirements far exceed the storage capacity of one node.* One alternative is a traditional parallel file system such as Lustre [21]. A second alternative is existing KVS systems, such as Redis [5] or Memcached [3].

Scalability. Consistent with our goal for high performance is the goal of high scalability on both HPC architectures and HPC applications. In this regard, PapyrusKV must scale to perhaps millions of application threads, petabytes of data, and architectures with high performance interconnection networks and massive amounts of aggregate memory including both DRAM and NVM.

Interoperability with existing programming models. PapyrusKV must be designed so that it can be incrementally introduced into an application without conflicting with existing HPC programming models and languages like MPI, UPC, OpenMP, OpenACC, C, C++, and Fortran. Furthermore, PapyrusKV should leverage characteristics of these other programming models when possible (e.g., the fact that many HPC applications must already coordinate execution, data movement, and synchronization). PapyrusKV should leverage these characteristics, providing only the minimal functionality necessary for maintaining KVS.

Application customizability. HPC applications have many different usage scenarios, and thus PapyrusKV should have customizable parameters for key features that impact other important properties like performance and scalability. For example, we allow the application to dictate the memory consistency model and protection attributes, such as read-only, on PapyrusKV data. This flexibility provides opportunities for improving scalability and performance.

2.2 Overview of PapyrusKV

PapyrusKV is a parallel KVS providing scalable high-performance data management to HPC applications. Figure 1 is a high-level overview of the PapyrusKV architecture. PapyrusKV stores keys and their values in arbitrary byte arrays across multiple NVM devices in a distributed system. It is embedded in SPMD- or MPMD-style HPC applications such as MPI and PGAS. PapyrusKV runs in a distributed fashion without any centralized control that may

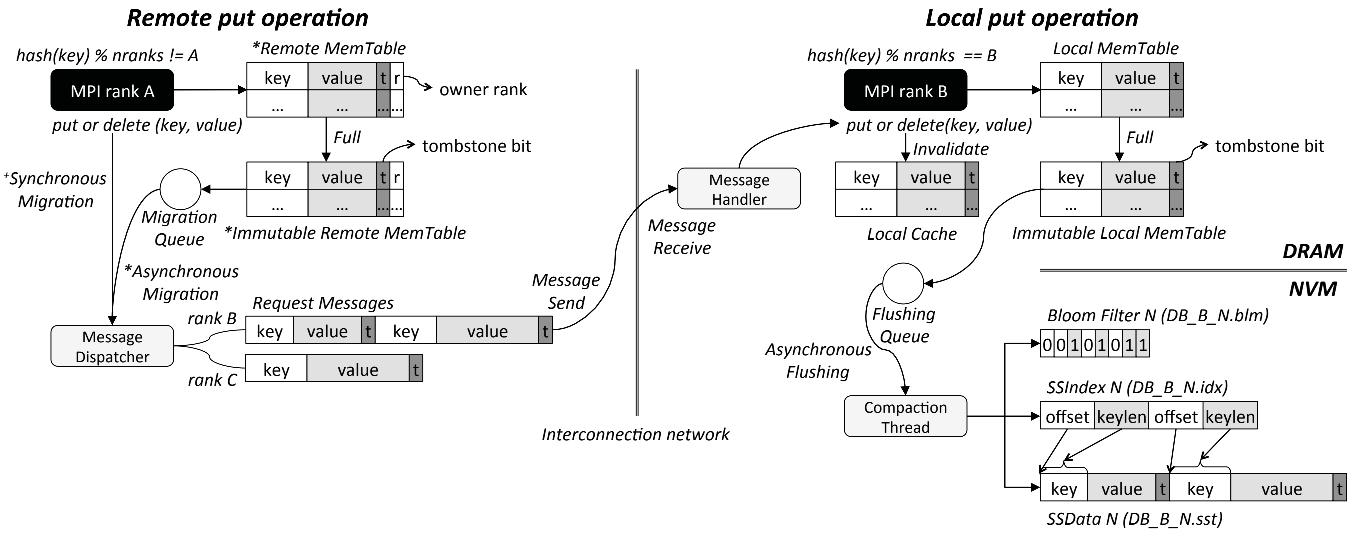

Figure 2: Put operations. *Remote MemTable and asynchronous migration in remote put operations are enabled when the database is set to the relaxed consistency mode. +In the sequential consistency mode, a single put operation is sent to the remote owner rank synchronously and directly without staging in the remote MemTable.

become a significant bottleneck in large-scale systems [16]. Since PapyrusKV is implemented as a user-level library using MPI, and it does not require any type of system-level daemon or server, it runs on most existing HPC systems [24].

Table 1 describes the PapyrusKV API functions. Every function returns a 32-bit integer error code such as `PAPYRUSKV_SUCCESS`, `PAPYRUSKV_INVALID_DB`, `PAPYRUSKV_NOT_FOUND`, etc. The PapyrusKV API provides basic operations such as opening and closing a database; reading, writing, and deleting a key-value pair; and more advanced features designed for HPC such as dynamic consistency control, zero-copy workflow, and asynchronous checkpoint/restart.

2.3 Opening a Database

A database can be opened or created using `papyruskv_open()` with a name. `papyruskv_open()` returns an associated database descriptor in `papyruskv_db_t` type. It is a collective operation that all participating MPI ranks need to call. Every MPI rank will have the same identical descriptor for the database.

A database consists of four types of MemTables (*local MemTable*, *immutable local MemTable*, *remote MemTable*, and *immutable remote MemTable*), two caches (*local cache* and *remote cache*), and a set of *SSTables* as shown in Figure 2 and Figure 3. The MemTable [20] is an in-memory data structure holding multiple key-value pairs. When a MemTable's size reaches its capacity limit, it is flushed to an SSTable [20] or other MPI ranks depending on its type. SSTable stands for Sorted String Table, which stores a set of immutable key-value pairs in sorted order based on the keys. It is stored on NVM storages and not written to again after the MemTable is flushed. The PapyrusKV runtime accesses the NVM storages through the standard POSIX file system interface [7].

The cache is a kind of MemTable, and it is managed in a LRU fashion. The local and remote caches store key-value pairs fetched from SSTables and other remote MPI ranks, respectively. Programmers can configure the database properties (e.g., MemTable capacity, cache on/off, cache capacity, memory consistency mode, protection attribute, and custom hash function). Multiple databases can be opened in a single application at a time, and they can have different properties.

2.4 Inserting Key-Value Pairs

A key-value pair is inserted or updated in the database using `papyruskv_put()`. Figure 2 illustrates how PapyrusKV handles put operations. PapyrusKV uses a hash algorithm to determine where the key-value pair is stored. It hashes the key and divides the result by the total number of the running MPI ranks. Then PapyrusKV uses the remainder to map the key and the *owner* MPI rank that stores the key-value pair in its memory or NVM storage. When the owner MPI rank is the same as the caller MPI rank, we call it a *local put operation*; the PapyrusKV runtime inserts the key-value pair to the local MemTable. If another key-value pair that has the same key already exists in the local MemTable, then PapyrusKV deletes the old one before it inserts the new one. Also, a stale cache entry that has the same key as the new key-value pair is evicted from the local cache. Each key-value pair in the local MemTable consists of a key, value, and tombstone bit flag. The MemTable is implemented as a red-black tree indexed by key. A red-black tree is a self-balancing binary tree. Thus, insert, lookup, and delete operations take $O(\log n)$ time.

Flushing. When the local MemTable's size reaches its capacity limit, it becomes an *immutable local MemTable*, and a new MemTable is created to handle new writes. The PapyrusKV runtime enqueues the immutable local MemTable to the *flushing queue*. The flushing queue is a lock-free, fixed-size, FIFO queue. After enqueuing, the caller MPI rank continues its execution and the runtime wakes up the *compaction thread*. The compaction thread is a background thread that dequeues the enqueued immutable MemTable from the queue and flushes it to a new SSTable on the NVM storage. The flushed immutable local MemTable is freed from the memory.

Because the flushing is executed by the compaction thread asynchronously with the MPI rank, the MPI rank does not need to wait for the completion of flushing. If the flushing queue is full when the runtime enqueues an immutable local MemTable into the queue, the MPI rank is blocked on the put operation until the queue is available. This prevents the unflushed MemTables from consuming too much system memory due to the performance imbalance between DRAM and NVM.

An SSTable is created for each local MemTable by the compaction thread, and it is immutable (i.e., not written to again). Each SSTable is assigned a per-database, per-rank, unique increasing integer number starting with one. We call the number *SSID*. An SSTable consists of three files, *SSData*, *SSIndex*, and *bloom filter*. SSData contains the actual key-value pair data. The stored data are sorted by key. SSIndex stores the offsets and lengths of keys of the key-value pairs in SSData. Bloom filter is a bit vector used to test whether an element is a member of a set [8]. Given an arbitrary key, it identifies whether the key may exist or definitely does not exist in the SSData. The bloom filter increases the probability of a successful lookup.

Migration. When the MPI rank executes a *remote put operation* (i.e., the owner of a new key-value pair is not the same as the caller MPI rank), the runtime inserts the key-value pair to the remote MemTable. Unlike the local MemTable, each key-value pair in the remote MemTable contains its owner rank number. When the remote MemTable's size reaches its capacity limit, it becomes an *immutable remote MemTable* and enqueued to the *migration queue*. And then, the caller MPI rank continues its execution and the runtime wakes up *message dispatcher thread*, called message dispatcher in short.

The message dispatcher dequeues the immutable MemTable from the migration queue. First, it sorts the key-value pairs in the MemTable by the owner rank number. Then the message dispatcher accumulates the key-value pairs per rank. It generates a number of request messages that contain a chunk of key-value pairs per rank and sends them to the target owner ranks via interconnection network.

In each target rank, *message handler thread*, called message handler in short, receives the request messages from the source rank. The message handler extracts the keys and their values from the messages and inserts them into the local MemTable. We call this *migration* to distinguish it from flushing in the local put operations. Like flushing, migration is also asynchronous with caller MPI rank. Furthermore, it is executed without remote MPI ranks' intervention. To achieve the interoperability between the user MPI applications and PapyrusKV runtime that uses MPI internally, the runtime creates new independent MPI communicators and uses them in the message dispatcher and message handler.

Load balancing. PapyrusKV relies on a hash function that determines the owner rank for a given key to provide an effective balance without additional metadata. However, a single hash function alone is not able to achieve efficient load balancing for all cases [22]. If the hash function does not uniformly distribute the key-value pairs across the running MPI ranks, communication and storage load imbalances occur among the compute nodes. This can harm the processing latency, system throughput, and storage availability.

To handle the load imbalance problem, PapyrusKV lets users customize the hash function in the PapyrusKV runtime. A custom hash function written by the user can be specified in a function parameter, `papyruskv_option_t` in `papyruskv_open()`, when a database is created, as shown in Table 1. The PapyrusKV runtime, which is embedded in the user application, internally calls the specified custom hash function to determine where the key-value pair in the database is stored. If the custom hash function is not specified by the user, the runtime uses its own built-in hash function.

2.5 Deleting Key-Value Pairs

A key-value pair with a given key is deleted from the database using `papyruskv_delete()`. The PapyrusKV runtime regards a delete operation as a put operation with zero-length value and a *tombstone* bit [20] set to one. The tombstone represents a key-value pair that has been deleted from a database.

Compaction. Once an SSTable is created, it is immutable. Updates and deletes cannot touch the data in the SSTable. The new data in updates and deletes are stored in new SSTables with higher SSIDs. Therefore, as updates and deletes occur, instead of overwriting the data, the number of SSTables increases, resulting in waste of NVM storage with stale key-value pairs. To solve this problem, PapyrusKV merges the data in a set of SSTables by the compaction thread whenever the SSID (per-database, per-rank, unique ID) of a new SSTable is multiples of the predefined number. It is called *compaction* [20]. The key-value pairs in the new merged SSTable are also sorted by key. During the compaction, if there are multiple key-value pairs with the same key, the key-value pair in the newest SSTable that has the highest SSID is inserted in the new merged SSTable. When the compaction is finished, the old SSTables are deleted to save storage space. The compaction needs sequential file read because the key-value pairs in each SSTable are sorted by the key.

2.6 Retrieving Key-Value Pairs

A key-value pair for a given key can be retrieved from the database using `papyruskv_get()`. Figure 3 illustrates how PapyrusKV handles get operations. Like put and delete operations, PapyrusKV hashes the given key to obtain the key-value pair's owner rank. If it is a local get operation, the PapyrusKV runtime checks the local MemTable first. When the runtime finds the matched key-value pair, it stops searching. If the found key-value pair is not a tombstone, PapyrusKV returns the found value to the application with `PAPYRUSKV_SUCCESS` code. Otherwise, it returns NULL with `PAPYRUSKV_NOT_FOUND` error code.

If the PapyrusKV runtime fails to find the matched key-value pair in the local MemTable, it checks the immutable local MemTables in the flushing queue with the newest first (i.e., from the tail to the head). If the search fails again, it checks the local cache. The local cache contains the key-value pairs fetched from SSTables. If the matched key-value pair is not found in the local cache again, the runtime walks the sequence of SSTables on the NVM storage until it finds the matched one. The PapyrusKV runtime checks the SSTables

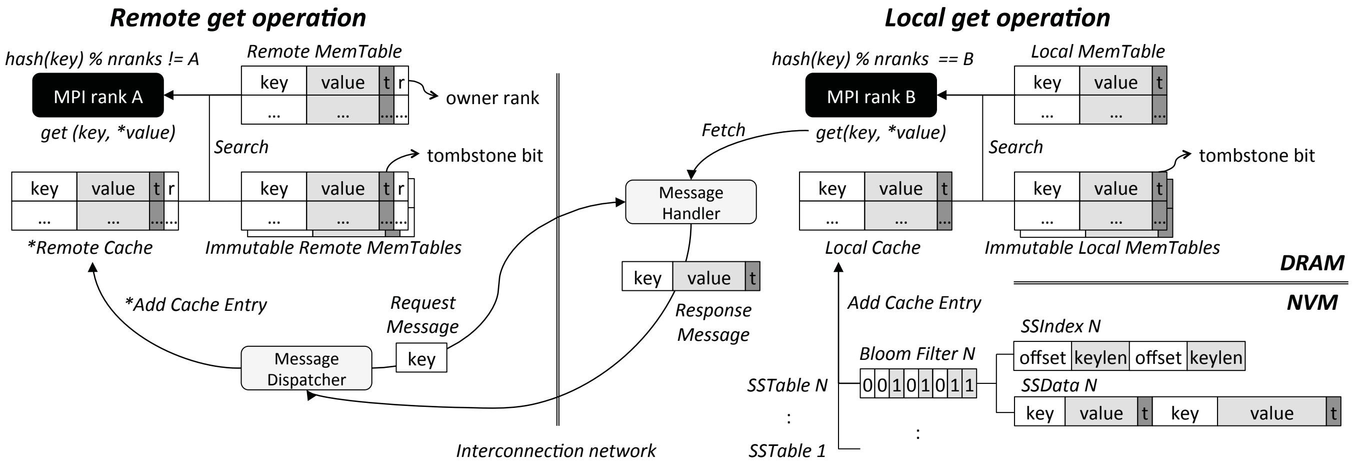

Figure 3: Get operations. *Remote cache is enabled when the database is write protected (i.e., read-only).

with the highest SSID first because more recent key-value pairs are stored in the SSTables with higher SSIDs. PapyrusKV opens the bloom filter file in the SSTable first to determine whether the SSTable can be skipped. When the bloom filter matches the given key, PapyrusKV opens the SSIndex and SSData files.

Binary search. PapyrusKV loads the SSIndex in memory and searches SSData with the given key. The key-value pairs in SSData are sorted by the key. NVMs show extraordinarily high performance for random access unlike traditional HDDs [11]. PapyrusKV looks into the SSData using a binary search algorithm. With n elements, the sequential search requires $O(n)$ time on average to find the matched key-value pair, whereas the binary search needs $O(\log n)$ time even in the worst case. When PapyrusKV finds the matched key-value pair in an SSData file, it inserts the found key-value pair to the local cache. The inserted cache entry will be invalidated when a new key-value pair with the same key is inserted in the local MemTable later as shown in Figure 2.

For a remote get operation, the PapyrusKV runtime looks into the remote MemTable, immutable remote MemTables in the migration queue, and remote cache in order. When it fails to find the matched key-value pair from them, the runtime creates a request message that contains the key information. The message dispatcher sends the message to the remote owner rank via the interconnection network and waits for the response message from the owner rank. The request message is received by the message handler in the owner rank. With the key information in the received message, the message handler performs a local get operation. And then the message handler sends the response message that contains the searching result to the caller rank. The message dispatcher in the caller rank receives the response message, inserts the received key-value pair in the remote cache, and returns the result to the application.

2.7 Storage Group

PapyrusKV introduces *storage group* to reduce the communication overhead between MPI ranks in the distributed NVM architecture. A storage group consists of one or more MPI ranks. All MPI ranks in the same storage group share NVM storages. They can access (read-only) others' SSTables stored in the shared NVM storage directly as shown in Figure 4. Each storage group is assigned a unique *storage group ID*.

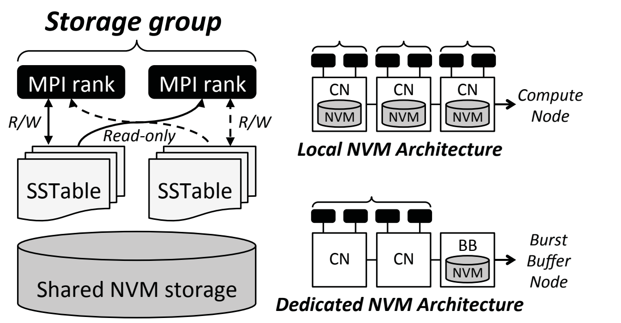

Figure 4: Storage group. All MPI ranks in the same storage group share the SSTables stored in shared NVM storage.

We categorize the distributed NVM architecture into *local NVM architecture* and *dedicated NVM architecture*. In the local NVM architecture, each compute node is equipped with one or more local NVM devices. The NVM devices are private to the local node and cannot be accessed directly by remote compute nodes. OLCF's Summit, ALCF's Theta, and LLNL's Sierra will have the local NVM architecture. In the local NVM architecture, all running MPI ranks in the same node compose a single storage group.

In the dedicated NVM architecture, on the other hand, the NVM devices reside on the dedicated storage nodes, typically called *burst buffer* nodes, connected with compute nodes. The NVM devices in the burst buffer nodes are accessible from all compute nodes in the system. NERSC's Cori has and LANL/Sandia's Trinity will have the dedicated NVM architecture. Typically, they conform to the POSIX consistency semantics that requires essentially sequential consistency of I/O operations [7]. In the dedicated NVM architecture, all running MPI ranks compose a single storage group.

In the remote get operation, the request message contains the caller rank's storage group ID. The message handler in the owner rank checks whether the storage group ID of the caller rank is the same as the owner rank's one. If they are same, that is, they share the NVM storages, the message handler in the owner rank looks into the local MemTable, immutable local MemTables, and local cache only. If the matched key-value pair is not found, the message handler sends back a response message to the caller rank that contains the results, the storage group ID, and the most recently flushed SSTable's SSID. The message dispatcher in the caller rank is notified that the key-value pair is not in the memory on the owner rank. However, it may reside in the SSTables on the shared NVM storage that can be read by the caller rank. The message dispatcher looks into the shared owner's SSTables in the shared NVM storage as if it were a local get operation. This mechanism can eliminate the unnecessary communication overhead to transfer the key-value pair data stored on the shared NVM storage between different ranks in the same storage group.

3 CONFIGURABLE CONSISTENCY

3.1 Relaxed vs. Sequential Consistency

PapyrusKV supports two memory consistency modes, *relaxed* and *sequential*. The memory consistency mode in PapyrusKV constrains the order in which key-value operations become visible to MPI ranks. Different databases can have different consistency modes at a time. The consistency mode in a database is specified when the database is created. Also, it can be changed dynamically during program execution by a collective function `papyruskv_consistency()`. Thus, the programmer can configure the memory consistency of the database by considering the application's behavior during the whole execution time or a specific time frame.

In the relaxed consistency mode, the key-value data in a database visible to different MPI ranks may be different except at synchronization points. For example, as shown in Figure 2, a remote put operation in the MPI rank A updates its remote MemTable only. The other ranks will retrieve NULL or a stale key-value pair from the owner rank B before the new key-value pair in rank A is migrated to the owner rank B.

PapyrusKV introduces two synchronization primitives, *fence* and *barrier*, used in the relaxed consistency mode. When an MPI rank runs a fence operation by calling `papyruskv_fence()`, the PapyrusKV runtime immediately migrates the remote MemTable and all immutable remote MemTables in the migration queue to the remote owner ranks. Barrier is a collective fence operation. It needs global synchronization that causes an MPI rank to halt execution until all the other MPI ranks have executed their corresponding barriers. Therefore, after the application calls `papyruskv_barrier()`, it is guaranteed that all MPI ranks will see the same latest data in the database. `papyruskv_barrier()` has a `level` parameter. When the parameter is set to `PAPYRUSKV_SSTABLE`, the PapyrusKV runtime forces all MPI ranks to flush their local MemTables and immutable local MemTables to the SSTables after they have received key-value pairs from all other ranks.

When a database is configured with the sequential consistency mode, a single remote put or delete operation is migrated to the remote owner rank immediately and synchronously without staging in the remote MemTable, as shown in Figure 2. The message dispatcher sends a request message to the remote owner rank whenever a put or delete operation is called. The caller MPI rank halts its execution until the message dispatcher is notified of the completion of migration from the owner rank. Therefore, every single put or delete operation becomes a synchronization point in the sequential consistency mode. The programmer can make the synchronization points order among the MPI ranks by using *signal* primitive. PapyrusKV presents two signal API functions: `papyruskv_signal_notify()` and `papyruskv_signal_wait()`.

3.2 Protection Attributes

If an application can be divided into write-only and read-only phases (or it has only one of them), the programmer can improve the application performance with help from the protection attribute configuration. The protection attribute of a database can be changed dynamically using `papyruskv_protection()` during program execution.

In the write-only phases, the database can be protected with `PAPYRUSKV_WRONLY`. The runtime invalidates all cache entries in the local cache and disables it. Therefore, PapyrusKV runtime does not need to invalidate the corresponding cache entry when a new key-value pair is inserted. This can save CPU cycles and memory bandwidth.

In the read-only phases, the database can be protected with `PAPYRUSKV_RDONLY`. The runtime enables the remote cache. As shown in Figure 3, when the message dispatcher receives a response message from a remote owner MPI rank, it creates a new corresponding key-value pair and inserts it in the remote cache. Because the database is write protected, the inserted key-value pair in the remote cache is always valid until the database is set to writable (`PAPYRUSKV_RDWR` or `PAPYRUSKV_WRONLY`). Therefore, the runtime looks into the remote cache first before it sends a request message to the remote owner rank. If the cache is hit, it can eliminate the communication and file I/O overhead to transfer the key-value pair data. All the key-value pairs in the remote cache are evicted, and the remote cache becomes disabled when the database is set to writable.

4 PERSISTENCE SUPPORT

Typically, as in OLCF's Summit, TACC's Stampede, and NERSC's Cori, NVM storages in HPC systems are used as a scratch space during a single job run because they are shared resources between multiple users and must adhere to traditional security and resource management policies. PapyrusKV is constrained by the architecture, system software, and storage policy of the systems. Figure 5 illustrates how PapyrusKV exploits NVM's persistence property in three different scenarios.

4.1 Zero-Copy Workflow

Because PapyrusKV stores SSTables on NVM storages, the lifetime of SSTables is beyond that of the application. SSTables are persistent in NVM devices during a single job. They are distributed across all the participating MPI ranks using a hash algorithm to determine the owners of key-value pairs. Therefore, as shown in

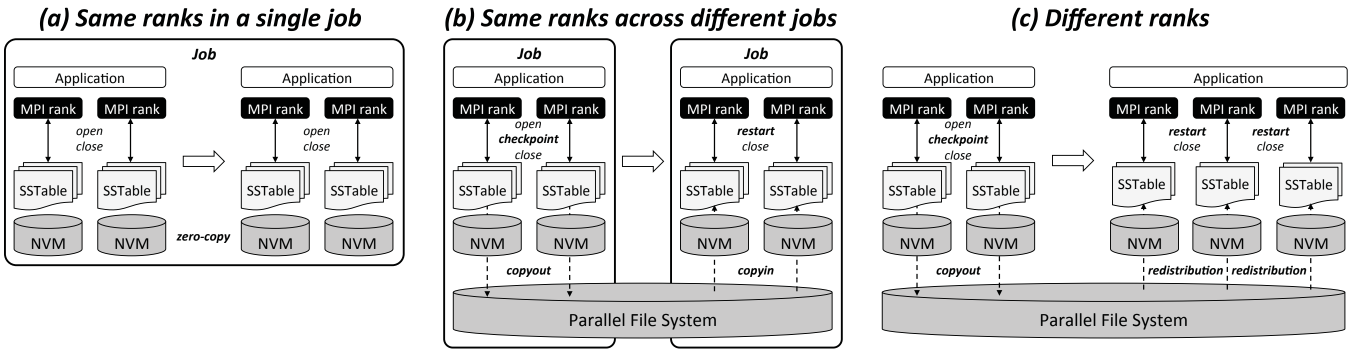

Figure 5: Zero-copy workflow and checkpoint/restart.

Figure 5(a), PapyrusKV databases can be reused across multiple application executions in a single job if they have the same number of ranks. This improves the workflow performance between a set of coupling applications using PapyrusKV as an interim storage. The latter applications can access the database created by the former applications by calling the `papyruskv_open()` API function with the same database name. The PapyrusKV runtime composes the database using the SSTables retained on the NVM storages. The new database consists of empty MemTables and a set of SSTables. This does not require any communication and file I/O (i.e., *zero-copy*).

In a dedicated NVM architecture such as NERSC's Cori, *persistent reservation* is available. It creates a space on the burst buffer nodes that will remain persistent after the end of a job. Therefore, multiple jobs that need the same database can also exploit the zero-copy workflow provided by PapyrusKV.

4.2 Checkpoint/Restart

Because NVM storages are trimmed after a job completes, the SSTables stored on the NVMs cannot be shared across multiple applications in different jobs. Thus, the SSTables need to be copied back to a parallel file system such as Lustre or GPFS before the job ends.

Checkpoint. In the HPC context, *checkpoint/restart* is one of the most popular fault tolerance mechanisms [9]. PapyrusKV provides asynchronous checkpoint/restart functionality to HPC applications. A collective function `papyruskv_checkpoint()` generates a snapshot image of the database. It is executed asynchronously with the MPI ranks. When `papyruskv_checkpoint()` is called, the runtime internally calls `papyruskv_barrier()` with the `PAPYRUSKV_SSTABLE` parameter. All the MemTables in all running MPI ranks are flushed into the SSTables. That is, a snapshot image of the database is created on the NVM storages. After that, the MPI ranks continue their executions, and the compaction thread in each rank starts to transfer the SSTables from NVM to the target parallel file system. While the compaction thread runs, the MPI rank is free to update the database because updates do not touch the existing SSTables in the snapshot. `papyruskv_checkpoint()` returns an event handle in the `papyruskv_event_t` type that identifies the asynchronous pending operation. The event handle can be used in `papyruskv_wait()`, which waits until the pending operation completes.

Restart. In case of failure, the application can restart with the snapshot image on the parallel file system using `papyruskv_restart()`. Also, the PapyrusKV restart operation can be used for streamlined workflow. As shown in Figure 5(b), if the number of ranks in the coupled applications in different jobs are the same, then the SSTables in the snapshot can be reused as they are, without any additional file manipulation. When MPI ranks call `papyruskv_restart()`, the compaction thread in every rank starts to transfer the SSTables in the parallel file system to the NVM storages. When the file transfers complete, the runtime internally calls `papyruskv_open()` to compose the database. Like the checkpoint, restart is executed asynchronously with the MPI rank.

Restart with redistribution. In PapyrusKV, a hash function (either built-in or custom) determines the owner rank of the given key using the total number of ranks. Therefore, if the number of ranks in the coupled applications are different, the SSTables in the snapshot cannot be reused as they are. In this case, the PapyrusKV runtime redistributes the key-value pairs in the SSTables across the running MPI ranks as shown in Figure 5(c). The compaction thread in each MPI rank reads the SSTables from the parallel file system, and calls a put operation for every key-value pair in the SSTables. The workload of put operations is partitioned across all the MPI ranks and executed in parallel.

5 EVALUATION

5.1 Methodology

We evaluated the performance of PapyrusKV using three HPC systems equipped with NVMs: OLCF's Summitdev, TACC's Stampede, and NERSC's Cori. Summitdev and Stampede have local NVM architectures. Summitdev is an early access test and development system for OLCF's next generation supercomputer, Summit. Each node in Summitdev is equipped with a node-local 800GB NVMe storage. In Stampede, each node includes 112GB of local SSD. Cori has a dedicated NVM architecture. SSDs are installed in burst buffer nodes. Each burst buffer node contains two 3.2TB NAND flash SSD modules. The compute nodes access the 1.8PB aggregate SSDs on the burst buffer nodes via the interconnection network. Table 2 summarizes the target systems.

Table 2: The target HPC systems.

System	Summitdev	Stampede (KNL)	Cori (Haswell)
Site	OLCF	TACC	NERSC
NVM architecture	Local NVM architecture, 800GB NVMe per compute node	Local NVM architecture, 112GB SSD per compute node	Dedicated NVM architecture, 1.8PB aggregate SSDs, 6.4TB SSDs per burst buffer node
Compute nodes	54	508	2,004
CPUs	2 × IBM POWER8	Intel Xeon Phi 7250 (KNL)	2 × Intel Xeon E5-2698 (Haswell)
Cores per CPU	10	68	16
HW threads per CPU	80	272	32
CPU core clock	2.0GHz	1.40GHz	2.30GHz
Main memory	256GB DDR4	96GB DDR4	128GB DDR4
Interconnection	Mellanox InfiniBand EDR	Intel Omni-Path	Cray Dragonfly interconnect
File systems	Local NVMe, Lustre	Local SSD, Lustre	SSD Burst Buffer nodes, Lustre, GPFS
OS	Red Hat Enterprise Linux Server 7.3	CentOS 7.3	SUSE Linux Enterprise Server 12
C/C++ compiler	IBM XL C/C++ 13.1.5	Intel C Compiler 17.0	Intel C Compiler 17.0.2
MPI	IBM Spectrum MPI 10.1.0	Intel MPI Library 2017	Cray MPICH 7.4.4
UPC	N/A	N/A	Berkeley UPC 2.24.2

5.2 Results

NVM performance. To show the effectiveness of using NVM, we measured the performance of the basic operations (put, barrier, and get) on various types of storages, such as local NVMe, local SSD, SSD burst buffer, and Lustre. Figure 6 shows the results. Multiple MPI ranks in a single node perform the operations on a database with the relaxed consistency mode in parallel. A total of 20, 68, and 32 MPI ranks are used in Summitdev, Stampede, and Cori, respectively. Each number is the same as the number of active physical CPU cores per node in each system. Enlarging it to the number of active hardware threads is restricted by the system resource manager (Summitdev and Cori) or not recommended (Stampede). The MemTable threshold value is set to 1GB. A rank performs each type of operations 10K times (Summitdev and Cori) or 1K times (Stampede, due to its SSD capacity) with 16B keys and different sizes of values ranging in size from 256B to 1MB. The keys are random strings containing letters (a-Z) and digits (0-9). They are generated in a uniformly distributed manner.

A put operation in the relaxed consistency mode performs on the memory only. The flushing and migration are executed on background and hidden from the application. Therefore, its performance is dominated by main memory performance. For small-size values, it is bottlenecked by the slow random access latency of DDR4 [23]. As the values increase in size, the bandwidth of parallel operations increases due to the fast sequential access of DDR4 until it reaches the memory's bandwidth limit.

The barrier graphs in Figure 6 show the performance when the application calls `papyruskv_barrier()` with the `PAPYRUSKV_SSTABLE` parameter after the previous put operations complete. It guarantees all MemTables are flushed to SSTables in the local or remote NVMs. Both in Summitdev and Stampede, barrier operations show better performance than Lustre with small-size

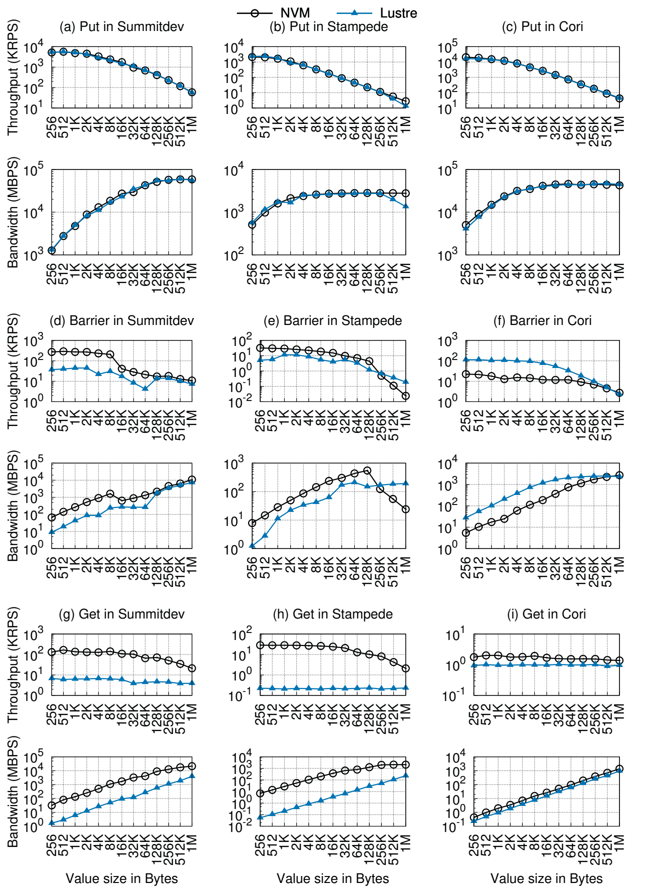

Figure 6: Basic operations performance in a single node. 20 (Summitdev), 68 (Stampede), and 32 (Cori) MPI ranks run the operations in parallel. We use KRPS (kilo requests per second) and MBPS (megabyte per second) as the metrics.

values but similar or poor performance with large-size values. This is because a file in the Lustre file system is striped across one or more object storage targets (OSTs), and writing to multiple OSTs simultaneously increases the available I/O throughput [21]. On the other hand, the results for Cori contrast with those for Summitdev and Stampede. In Cori, a file is also striped across several burst buffer nodes like Lustre. This enables parallel file I/O across multiple burst buffer nodes and delivers high bandwidth.

In get operations, NVMs show better performance than Lustre due to their fast read performance and extraordinarily high performance for random access. Local NVM architectures, Summitdev and Stampede, show orders-of-magnitude performance improvement over Lustre.

Relaxed vs. sequential consistency. Figure 7 shows the put operation performance in the relaxed (Rel) and sequential consistency modes (Seq). Each rank performs 10K (Summitdev and Cori) or 1K (Stampede) put operations with 16B keys and 128KB values. We vary the number of ranks from one to the same number of physical CPU cores in a node for each system and multiples of

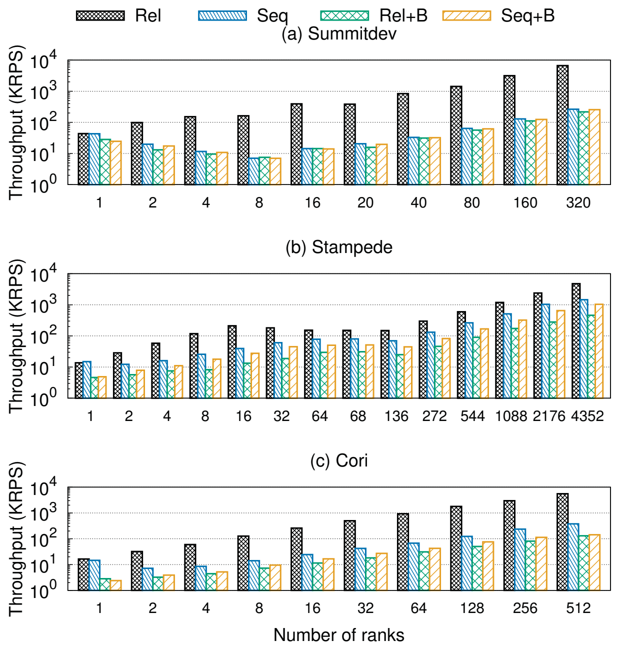

Figure 7: Put operation performance in relaxed (Rel) and sequential (Seq) consistency modes. B refers to barrier.

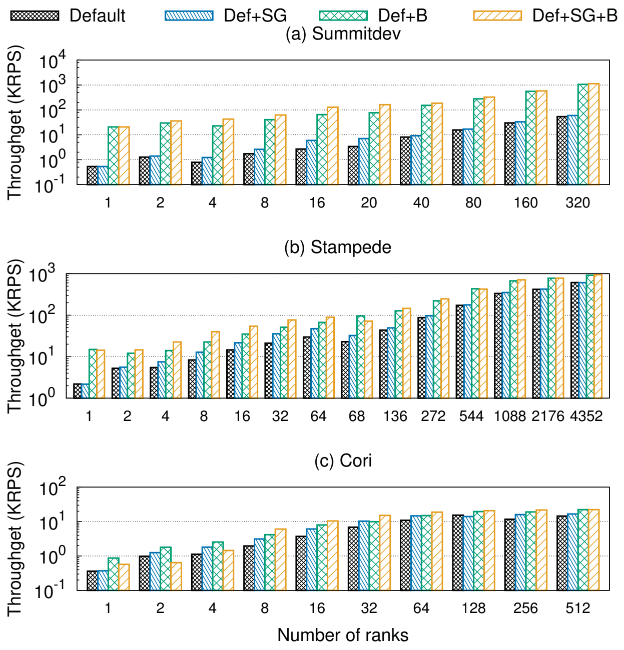

Figure 8: Get operation performance. SG and B refer to storage group and SSTable binary search, respectively.

them in a power of two. Because the keys are randomly generated, the put operations are mixed with local and remote operations.

The throughput of put operations in the relaxed mode shows higher performance than that in the sequential mode. This is because both of local and remote put operations in the relaxed mode update the main memory only while a remote put operation in the sequential consistency mode is migrated to the owner rank immediately and synchronously.

Figure 7 also shows the performance of barrier operations. The application calls a barrier operation when it finishes the put operations. We measure throughputs of them together in the relaxed (Rel+B) and sequential mode (Seq+B). Contrary to put operations, the sequential mode shows slightly higher throughput than the relaxed mode. This is mainly because the network congestion in the relaxed mode is more severe than that in the sequential mode. The barrier is a collective operation that incurs an all-to-all communication among all the MPI ranks at a time.

Storage group and SSTable binary search. PapyrusKV introduces two optimization techniques for get operations; storage group and SSTable binary search. The MPI ranks in the same storage group can directly read other ranks' SSTable in the shared NVM storage without SSTable data transfer between the ranks. SSTable binary search reduces the I/O overhead for searching SSTables in NVM storages by exploiting NVM's fast random access.

Figure 8 shows the performance of get operations when the storage group (SG) and/or SSTable binary search (B) techniques are applied and when none is applied (Default or Def). The sizes of storage groups (a total number of ranks in a storage group) are set to the total number of ranks in a node for local NVM architecture and the total number of ranks in the application for dedicated NVM architecture. That is, they are set to 20, 68, and 512 in Summitdev, Stampede, and Cori, respectively. Both techniques are effective and show best performance when they are combined (Def+SG+B). Def+SG+B shows 7%, 2%, and 7% performance improvement over Def+B on 320 ranks in Summitdev, 4352 ranks in Stampede, and 512 ranks in Cori, respectively.

Various workloads and caching. We measured PapyrusKV performance with various workloads. The application consists of two phases: initialization phase and read/update phase. In the initialization phase, it performs 10K (Summitdev and Cori) or 1K (Stampede) put operations with 16B keys and 128KB values on a database. In the read/update phase, the application calls get and put operations on the database with the keys used in the initialization phase a total of 10K or 1K times. The database is configured to sequential consistency mode. We varied the read/update ratio to 50/50, 95/5 and 100/0.

Figure 9 shows the results. In Summitdev, PapyrusKV shows better throughput as read ratio increases. This is because get operations show better throughput than put operations in sequential mode as shown in Figure 7 and Figure 8. Stampede and Cori show contrasting results with Summitdev because of their slow get operation performance. Especially when a database is used for read-only for a certain timeframe, the programmer can improve the performance of remote get operations by calling `papyruskv_protect()` with the `PAPYRUSKV_RDONLY` parameter that enables the remote cache. As shown in Figure 9, when the application protects the database in read-only during the read/update phase (100/0+P), it shows better performance than without protection (100/0).

Checkpoint/restart. We measured checkpoint/restart performance using the three coupled microbenchmark applications. The

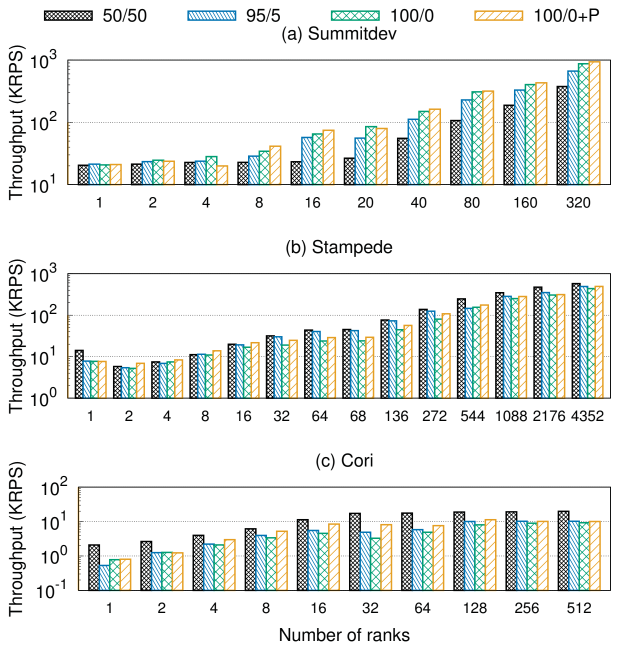

Figure 9: Various workloads. 50/50, 95/5, and 100/0 refer to read/update ratios in the application. P refers to protection that enables the remote cache.

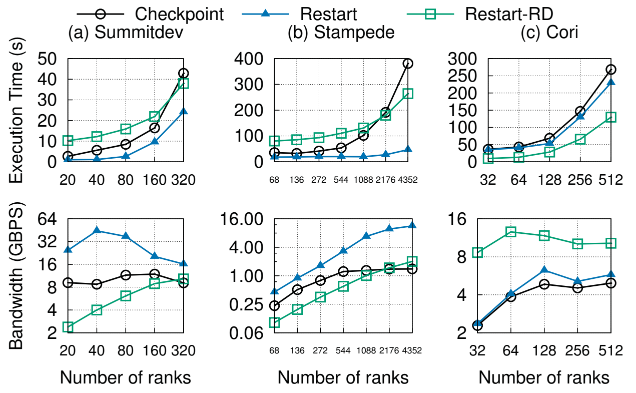

Figure 10: Checkpoint, restart, and restart with redistribution (RD) performance.

first application performs 10K (Summitdev and Cori) or 1K (Stampede) put operations with 16B keys and 128KB values, and then it calls a checkpoint operation that generates a snapshot of the database in Lustre. The second application reverts the database from Lustre using a restart operation. The last application reverts the database from Lustre through the restart with redistribution technique. All three applications run with the same number of ranks. Even though the last application does not need a redistribution, we forced it for the evaluation.

Figure 10 shows the total execution times of checkpoint, restart, and restart with redistribution (RD) and their bandwidths. The throughputs and bandwidths of checkpoint and restart are fully

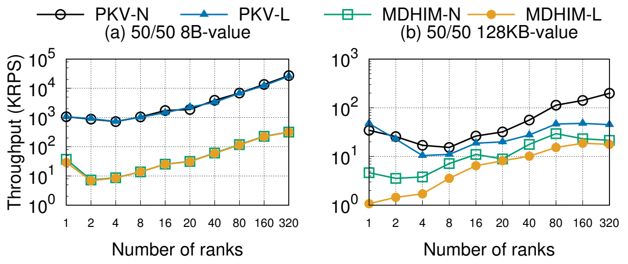

Figure 11: Performance comparisons between PapyrusKV (PKV) and MDHIM on Summitdev. NVMe (N) and Lustre (L) are used for their data storages.

dominated by the parallel I/O performance in the NVM storages and Lustre. On the other hand, the redistribution performance reflects parallel put operation performance as well as the parallel I/O performance in Lustre.

Comparisons with MDHIM. MDHIM [14] is another embedded and distributed KVS for HPC. It distributes key-value pairs across multiple running nodes. It presents a communication/distribution layer on top of the local data store such as LevelDB [2] and MySQL. Like PapyrusKV, MDHIM does not require any daemons and dedicated servers. We compared PapyrusKV performance with that of MDHIM.

Figure 11 shows the performance comparisons of PapyrusKV and MDHIM on Summitdev. We used LevelDB as the local data store of MDHIM. We measured their throughputs using the same application used to measure performance under various workloads (Figure 9). Each rank in the application runs a total of 10K put (update) and get (read) operations. The update/read ratio is set to 50/50. The size of key is 16B and we varied the size of value to 8B and 128KB. We used the local NVMe and Lustre as storage devices.

For 8B-value, PapyrusKV and MDHIM show almost the same performance between NVMe (N) and Lustre (L) because the size of the database is not big enough to be flushed to SSTables. The update/read operations execute on DRAM only. PapyrusKV outperforms MDHIM both in performance and scalability. This is mainly because MDHIM maintains two discrete memory data structures in the communication/distribution layer (MDHIM) and local data storage layer (LevelDB). It incurs additional duplicated memory allocation and data transfer between the two layers. PapyrusKV, on the other hand, is a single framework that tightly integrates the communication/distribution and data storage layers. The layers share the memory data structures, and it does not introduce any unnecessary memory allocation and data transfer between them, resulting in better performance.

For 128KB-value, the SSTables are created on the storages and the update/read operations run using them. Both PapyrusKV and MDHIM with NVMe show better performance than Lustre due to its lower latency and higher bandwidth. PapyrusKV, in addition to eliminating unnecessary overhead between the communication/distribution and data storage layers, eliminates the key-value pair data transfer between different ranks in the same storage group by sharing SSTables. PapyrusKV exploits the storage group and shows higher performance and scalability than MDHIM, while

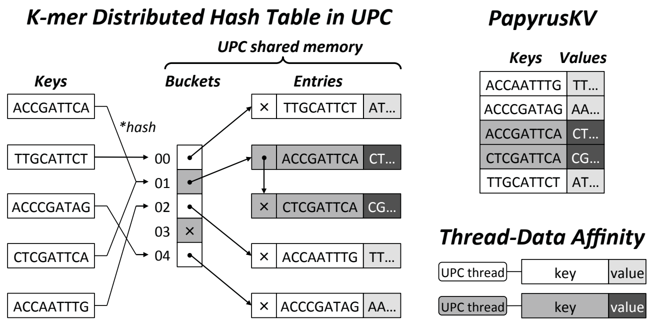

Figure 12: K-mer distributed hash table implementations in UPC and PapyrusKV. *The same hash function for load balancing in the UPC application is used in PapyrusKV.

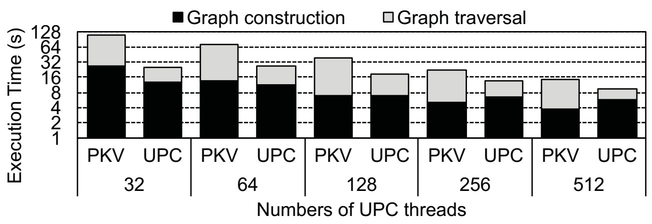

Figure 13: Meraculous performance comparison between PapyrusKV (PKV) and UPC on Cori.

MDHIM cannot share the SSTables between multiple independent LevelDB instances.

A real HPC application. Distributed shared-memory (DSM) programming, such as Unified Parallel C (UPC), Co-Array Fortran, SHMEM, and Global Arrays, simplifies parallel programming by presenting a single address space among multiple tasks on a distributed system. PapyrusKV can be regarded as a DSM programming model in the sense that it provides a shared and parallel data structure across multiple tasks in the distributed memory systems.

Meraculous [13] is a state-of-the-art de novo assembler written in UPC. Its parallel algorithm for de Bruijn graph construction and traversal leverages the one-sided communication in UPC to facilitate the requisite random access pattern in the global de Bruijn graph. The de Bruijn graph is implemented as a distributed hash table with an overlapping substring of length *k*, referred to as a *k-mer*, as key and a two-letter code [ACGT][ACGT] as value as shown in Figure 12. A hash function is used to define the affinities between UPC threads and hash table entries.

We ported the distributed hash table written in UPC to a PapyrusKV database. The keys in the database are k-mers and the values are two-letter codes. The PapyrusKV runtime calls the same hash function in the UPC application to determine the owners of key-value pairs in the database by specifying the custom hash function when the database is created. Thus, the thread-data affinities in UPC and PapyrusKV are the same as shown in Figure 12. PapyrusKV requires fewer lines of source code than UPC because it calls standard put and get API functions without implementing an application-specific algorithm for the distributed hash table construction and traversal.

Figure 13 shows the performance comparison between PapyrusKV and UPC of Meraculous on Cori. Both versions are built and run using Berkeley UPC, an MPI-interoperable UPC implementation. We measured the total execution time using the *human chr14* dataset [4] on 32, 64, 128, 256, and 512 UPC threads (32 UPC threads per node). UPC shows better performance than PapyrusKV due to its RDMA capability and built-in remote atomic operations during the graph traversal. The performance gap between UPC and PapyrusKV decreases as the number of UPC threads increases. On 512 UPC threads, PapyrusKV runs 1.5 times slower than UPC. This is mainly because of the asynchronous migration in PapyrusKV during the graph construction.

6 RELATED WORK

A number of surveys [19, 27] cover much related work. Here, we highlight specific systems closely related to PapyrusKV.

KVSs based on LSM-trees. Various state-of-the-art KVSs are based on LSM-trees [20]. LSM-trees achieve high write throughput from their hierarchy of indexes spanning across memory and disk. PapyrusKV also follows the main concepts in LSM-trees, such as MemTables in memory, immutable SSTables on persistent storage (i.e., NVM), tombstone, and compaction. LevelDB [2] and RocksDB [6] are two popular single-node KVSs based on LSM-trees. LevelDB optimizes write throughput by organizing data in multiple levels. RocksDB is an optimization version of LevelDB. It is designed for fast, low latency storage such as SSDs and achieves both high write and read throughputs. PapyrusKV also delivers high write and read performance by exploiting LSM-trees oriented for high write performance and NVMs that provide extraordinarily high-speed random access. Moreover, beyond a single node, PapyrusKV targets the distributed systems.

Bigtable [10], Cassandra [17], and HBase [1] are LSM-tree-based KVSs for distributed systems. They distribute data and workload across multiple servers and are designed to achieve high scalability in terms of storage and throughput. Also, they ensure high availability even in failure on cloud infrastructure by replication of data and commit logs. PapyrusKV, on the other hand, is designed and optimized for HPC. It is difficult or even impossible to run persistent daemons or servers on HPC systems where application executions are scheduled by the system resource manager such as PBS or SLURM. Therefore, PapyrusKV is embedded with the applications as a user-level library and runs without any type of system-level daemon or server. To achieve fault tolerance, PapyrusKV exploits the checkpoint/restart technique like typical HPC applications.

KVSs for HPC. Wang et al. [26] simulate KVS for various HPC service workloads and justify the effectiveness and usefulness of distributed KVS for HPC. Scalable Key/Value Store (SKV) [12] and MDHIM [14] present KVSs designed for HPC. SKV is a distributed key-value database software framework specially designed for the IBM BlueGene/Q (BG/Q) system connected with Blue Gene Active Storage (BGAS). It has a server-client architecture. Multiple SKV servers run on the BGAS I/O nodes equipped with flash storages. The SKV client interface is embodied in a user-level library for MPI applications. The clients run on the BG/Q compute nodes, and they communicate with the SKV servers by using TCP/IP or OFED RDMA. Unlike SKV, PapyrusKV is portable across diverse distributed NVM architectures including the next-generation DOE supercomputers. Furthermore, PapyrusKV is an embedded KVS that does not require any servers in the system.

MDHIM [14] is a parallel embedded KVS designed for HPC systems. As shown in §5, PapyrusKV achieves better performance and scalability than MDHIM by sharing SSTables between multiple ranks in the same storage group and tightly integrating the communication/distribution layer and local data store layer. Moreover, PapyrusKV presents configurable consistency to exploit the HPC applications' characteristics and supports persistence mechanism leveraging NVM.

7 CONCLUSIONS

This paper introduces PapyrusKV, a novel parallel KVS for distributed HPC architectures that offer potentially massive pools of NVM. It is designed to leverage emerging NVM technologies' high performance, high capacity, and persistence properties. In the HPC context, PapyrusKV is an embedded KVS implemented with MPI, and it meets HPC system resource management policies. Thus, PapyrusKV can run on most existing HPC systems. PapyrusKV provides advanced features specialized for HPC such as dynamic consistency control, zero-copy workflow, and asynchronous checkpoint/restart. Our evaluation of PapyrusKV on three HPC systems equipped with different NVM architectures demonstrates that PapyrusKV meets our key design goals: performance, scalability, interoperability with existing programming models, and application customizability.

ACKNOWLEDGMENTS

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC05-00OR22725 This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan.

REFERENCES

[1] *Apache HBase*. https://hbase.apache.org/.
[2] *LevelDB*. http://leveldb.org/.
[3] *Memcached*. https://memcached.org/.
[4] *Meraculous APEX Benchmark*. http://www.nersc.gov/research-and-development/apex/apex-benchmarks/meraculous/.
[5] *Redis*. https://redis.io/.
[6] *RocksDB*. http://rocksdb.org/.
[7] IEEE/ANSI Std. 1003.1. 1996. *Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]*.
[8] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM* 13 (1970), 422–426.
[9] K. Mani Chandy and Leslie Lamport. 1985. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 3, 1 (Feb. 1985), 63–75.
[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. 205–218.
[11] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*. 181–192.
[12] S. Eilemann, F. Delalondre, J. Bernard, J. Planas, F. Schuermann, J. Biddiscombe, C. Bekas, A. Curioni, B. Metzler, P. Kaltstein, P. Morjan, J. Fenkes, R. Bellofatto, L. Schneidenbach, T. J. C. Ward, and B. G. Fitch. 2016. Key/Value-Enabled Flash Memory for Complex Scientific Workflows with On-Line Analysis and Visualization. In *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS '16)*. 608–617.
[13] Evangelos Georganas, Aydin Buluç, Jarrod Chapman, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2014. Parallel De Bruijn Graph Construction and Traversal for De Novo Genome Assembly. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*. 437–448.
[14] Hugh N. Greenberg, John Bent, and Gary Grider. 2015. MDHIM: A Parallel Key/Value Framework for HPC. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'15)*.
[15] Jungwon Kim, Kittisak Sajjapongse, Seyong Lee, and Jeffrey S. Vetter. 2017. Design and Implementation of Papyrus: Parallel Aggregate Persistent Storage. In *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium (IPDPS '17)*. 1151–1162.
[16] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. 2012. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS '12)*. 341–352.
[17] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (2010), 35–40.
[18] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012. On the Role of Burst Buffers in Leadership-Class Storage Systems. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST '12)*.
[19] S. Mittal and J. S. Vetter. 2016. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1537–1550.
[20] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (1996), 351–385.
[21] Philip Schwan. 2003. Lustre: Building a File System for 1,000-node Clusters. In *Proceedings of the Linux Symposium*.
[22] Weiguang Shi, M. H. MacGregor, and Pawel Gburzynski. 2005. Load Balancing for Parallel Forwarding. *IEEE/ACM Trans. Netw.* 13, 4 (Aug. 2005), 790–801.
[23] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. 2008. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. 51–62.
[24] Jeffrey S. Vetter. 2013. *Contemporary High Performance Computing: From Petascale Toward Exascale*. Chapman & Hall/CRC.
[25] J. S. Vetter and S. Mittal. 2015. Opportunities for Nonvolatile Memory Systems in Extreme-Scale High-Performance Computing. *Computing in Science Engineering* 17, 2 (2015), 73–82.
[26] Ke Wang, Abhishek Kulkarni, Michael Lang, Dorian Arnold, and Ioan Raicu. 2013. Using Simulation to Explore Distributed Key-value Stores for Extreme-scale System Services. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. 9:1–9:12.
[27] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1920–1948.

A ARTIFACT DESCRIPTION: PAPYRUSKV: A HIGH-PERFORMANCE PARALLEL KEY-VALUE STORE FOR DISTRIBUTED NVM ARCHITECTURES

A.1 Abstract

This artifact contains all components of PapyrusKV presented in the SC 2017 paper titled PapyrusKV: A High-Performance Parallel Key-Value Store for Distributed NVM Architectures. It also includes scripts to run PapyrusKV on OLCF's Summitdev, TACC's Stampede (KNL), and NERSC's Cori (Haswell).

A.2 Description

A.2.1 Check-list (artifact meta information).

- **Algorithm:** Distributed Key-Value Store.
- **Program:** C/C++, MPI, and UPC.
- **Compilation:** IBM XL C/C++ 13.1.5 with IBM Spectrum MPI 10.1.0 in Summitdev, Intel C Compiler 17.0 with Intel MPI Library 2017 in Stampede, and Intel C Compiler 17.0.2 with Cray MPICH 7.4.4 and Berkeley UPC 2.24.2 in Cori.
- **Run-time environment:** Red Hat Enterprise Linux Server 7.3 in Summitdev, CentOS 7.3 in Stampede, and SUSE Linux Enterprise Server 12 in Cori.
- **Hardware:** OLCF's Summitdev, TACC's Stampede, and NERSC's Cori.
- **Output:** Average, minimum, and maximum of the total execution time.
- **Experiment workflow:** Download package, compile the source code and test applications, run the applications and check the outputs.
- **Publicly available?:** Yes.

A.2.2 How software can be obtained (if available). The git repository of PapyrusKV can be found at http://ft.ornl.gov/research/papyrus.

A.2.3 Hardware dependencies. Cluster systems equipped with NVMs.

A.2.4 Software dependencies. C++ compiler that supports C++11 standard. MPI library that supports a multi-threading level of MPI_THREAD_MULTIPLE. Berkeley UPC.

A.2.5 Datasets. Datasets are included with PapyrusKV package.

A.3 Installation

(1) Clone PapyrusKV and set an environment variable.

```
$ cd $HOME
$ git clone $PAPYRUSKV_GIT_REPOSITORY/papyrus.git
$ export PAPYRUS=$HOME/papyrus
$ export PAPYRUSKV=$PAPYRUS/kv
```

(2) Create an appropriate Makefile.machine in $PAPYRUS/conf directory. Sample Makefiles for Summitdev, Stampede, and Cori are included in the package. Makefile.machine must describe the compiler- and MPI-specific options.

(3) Build PapyrusKV libraries in $PAPYRUSKV/src directory.

```
$ cd $PAPYRUSKV/src
$ make
```

(4) Check libpapyruskv.a and libpapyruskv.so files are created in $PAPYRUS/lib directory.

```
$ ls $PAPYRUS/lib
libpapyruskv.a     libpapyruskv.so
```

A.4 Experiment workflow

(1) Build three microbenchmark applications (`basic`, `workload`, `cr`) in $PAPYRUSKV/apps/sc17 directory.

```
$ cd $PAPYRUSKV/apps/sc17
$ make
```

(2) `basic` application is used in Figure 6, Figure 7, and Figure 8. It takes three arguments as length of keys, length of values, and iteration count. The scripts used in Summitdev, Stampede, and Cori are included in the package. For example, you can run the application to reproduce the results in Figure 6 using Cori with the following script:

```
#SBATCH -J basic
#SBATCH -C haswell
#SBATCH -N 1
#SBATCH -t 01:00:00
#SBATCH -L SCRATCH
#DW jobdw capacity=1TB access_mode=striped type=scratch

export MPICH_MAX_THREAD_SAFETY=multiple
VALLEN=(256 512 1024 2048 4096 8192 16384 32768
        65536 131072 262144 524288 1048576)

# NVM in Figure 6.
export PAPYRUSKV_REPOSITORY=$DW_JOB_STRIPED/basic
for i in "$VALLEN[@]"; do
  srun -n 32 -c 2 ./basic 16 $i 10000
done

# Lustre in Figure 6.
export PAPYRUSKV_REPOSITORY=$SCRATCH/basic
for i in "$VALLEN[@]"; do
  srun -n 32 -c 2 ./basic 16 $i 10000
done
```

You can run the application to reproduce the results in Figure 7 and Figure 8 using Stampede with the following script:

```
#SBATCH -J basic
#SBATCH -N 64
#SBATCH -n 4352
#SBATCH -t 01:00:00
#SBATCH -A $YOUR_PROJECT

RANKS=(1 2 4 8 16 32 64 68 136 272 544 1088 2176 4352)
export PAPYRUSKV_REPOSITORY=/tmp/basic

# Seq, Seq+B in Figure 7.
export PAPYRUSKV_GROUP_SIZE=1
export PAPYRUSKV_CONSISTENCY=1
export PAPYRUSKV_BIN_SEARCH=1
for i in "$RANKS[@]"; do
  ibrun -np $i ./basic 16 131072 1000
done

# Rel, Rel+B in Figure 7.
# Default in Figure 8.
export PAPYRUSKV_CONSISTENCY=2
for i in "$RANKS[@]"; do
  ibrun -np $i ./basic 16 131072 1000
done

# Def+SG in Figure 8.
export PAPYRUSKV_GROUP_SIZE=68
for i in "$RANKS[@]"; do
  ibrun -np $i ./basic 16 131072 1000
done

# Def+B in Figure 8.
export PAPYRUSKV_GROUP_SIZE=1
export PAPYRUSKV_BIN_SEARCH=2
for i in "$RANKS[@]"; do
```

```
  ibrun -np $i ./basic 16 131072 1000
done

# Def+SG+B in Figure 8.
export PAPYRUSKV_GROUP_SIZE=68
for i in "$RANKS[@]"; do
  ibrun -np $i ./basic 16 131072 1000
done
```

(3) `workload` application is used in Figure 9 and Figure 11. It takes four arguments as length of keys, length of values, iteration count, and update ratio in percentage (0-100). The scripts used in Summitdev, Stampede, and Cori are included in the package. For example, you can run the application to reproduce the results in Figure 9 and Figure 11 using Summitdev with the following script:

```
#BSUB -J workload
#BSUB -W 1:00
#BSUB -n 320
#BSUB -env "all,JOB_FEATURE=NVME"
#BSUB -P $YOUR_PROJECT

RANKS=(1 2 4 8 16 20 40 80 160 320)
export PAPYRUSKV_GROUP_SIZE=20
export PAPYRUSKV_REPOSITORY=/xfs/scratch/$USERNAME/workload

# 50/50 in Figure 9. PKV-N in Figure 11.
for i in "$RANKS[@]"; do
  mpirun -np $i ./workload 16 8      10000 50
  mpirun -np $i ./workload 16 131072 10000 50
done

# 95/5 in Figure 9.
for i in "$RANKS[@]"; do
  mpirun -np $i ./workload 16 131072 10000 5
done

# 100/0 in Figure 9.
for i in "$RANKS[@]"; do
  mpirun -np $i ./workload 16 131072 10000 0
done

# 100/0+P in Figure 9.
export PAPYRUSKV_CACHE_REMOTE=1
for i in "$RANKS[@]"; do
  mpirun -np $i ./workload 16 131072 10000 0
done

# PKV-L in Figure 11.
export PAPYRUSKV_REPOSITORY=/lustre/atlas/scratch/$USERNAME/$PROJ/workload
for i in "$RANKS[@]"; do
  mpirun -np $i ./workload 16 8      10000 50
  mpirun -np $i ./workload 16 131072 10000 50
done
```

(4) `cr` application is used in Figure 10. It takes five arguments as length of keys, length of values, iteration count, lustre directory path, and checkpoint/restart flag (c/r). The scripts used in Summitdev, Stampede, and Cori are included in the package. For example, you can run the application to reproduce the results in Figure 10 using Cori with the following script:

```
#SBATCH -J cr
#SBATCH -C haswell
#SBATCH -N 16
#SBATCH -t 01:00:00
#SBATCH -L SCRATCH
#DW jobdw capacity=1TB access_mode=striped type=scratch

RANKS=(32 64 128 256 512)
export MPICH_MAX_THREAD_SAFETY=multiple
export PAPYRUSKV_GROUP_SIZE=512
export PAPYRUSKV_REPOSITORY=$DW_JOB_STRIPED/cr
export PAPYRUSKV_LUSTRE=$SCRATCH/cr

for i in "$RANKS[@]"; do
  # Checkpoint in Figure 10.
  srun -n $i -c 2 ./cr 16 131072 10000 $PAPYRUSKV_LUSTRE c

  # Restart in Figure 10.
  export PAPYRUSKV_FORCE_REDISTRIBUTE=0
  srun -n $i -c 2 ./cr 16 131072 10000 $PAPYRUSKV_LUSTRE r

  # Restart-RD in Figure 10.
  export PAPYRUSKV_FORCE_REDISTRIBUTE=1
  srun -n $i -c 2 ./cr 16 131072 10000 $PAPYRUSKV_LUSTRE r
done
```

(5) To reproduce the results in Figure 11, LevelDB and MDHIM are required. Download LevelDB from its Git repository (https://github.com/google/leveldb.git) and follow the build instruction in the package. Download MDHIM from its Git repository (https://github.com/mdhim/mdhim-tng.git) and follow the build instruction in the package. Then, copy mdhim.c in $PAPYRUSKV/apps/sc17 directory to $MDHIM/tests/single_tests directory and bulid `mdhim` application.

```
$ cp $PAPYRUSKV/apps/sc17/mdhim.c $MDHIM/tests/single_tests/
$ cd $MDHIM/tests/single_tests
$ echo -e "mdhim:mdhim.c\n\t\$(CC) \$< \$(CINC) \$(CLIBS)
          \$(CFLAGS) -o \$@" >> Makefile
$ make mdhim
```

You can run the application to reproduce the results in Figure 11 using Summitdev with the following script:

```
#BSUB -J mdhim
#BSUB -W 1:00
#BSUB -n 320
#BSUB -env "all,JOB_FEATURE=NVME"
#BSUB -P $YOUR_PROJECT

VALLEN=(8 131072)
RANKS=(1 2 4 8 16 20 40 80 160 320)

# MDHIM-N in Figure 11.
export PAPYRUSKV_REPOSITORY=/xfs/scratch/$USERNAME/mdhim
for i in "$RANKS[@]"; do
  for j in "$VALLEN[@]"; do
    mpirun -np $i ./mdhim 16 $j 10000 50
  done
done

# MDHIM-L in Figure 11.
export PAPYRUSKV_REPOSITORY=/lustre/atlas/scratch/$USERNAME/$PROJ/mdhim
for i in "$RANKS[@]"; do
  for j in "$VALLEN[@]"; do
    mpirun -np $i ./mdhim 16 $j 10000 50
  done
done
```

(6) To reproduce the results in Figure 13, build Meraculous in $PAPYRUSKV/apps/meraculous directory. It requires Berkeley UPC.

```
$ cd $PAPYRUSKV/apps/meraculous
$ ./build.sh
```

The input dataset file (`human-chr14.txt.ufx.bin`) and job scripts for Cori are included in the package.

```
$ sbatch run_pkv.pbs
$ sbatch run_upc.pbs
```

Verify the output contigs files.

```
$ cd tests
$ ./check_results.sh
```

A.5 Evaluation and expected result

The evaluation results can be found in the output log file. It contains the average, minimum, and maximum of total execution times for all MPI ranks in the application.