MDHIM: A Parallel Key/Value Framework for HPC

Hugh N. Greenberg
[1] *Los Alamos National Laboratory*

John Bent
EMC

Gary Grider
Los Alamos National Laboratory

Abstract

The long-expected convergence of High Performance Computing and Big Data Analytics is upon us. Unfortunately, the computing environments created for each workload are not necessarily conducive for the other. In this paper, we evaluate the ability of traditional high performance computing architectures to run big data analytics. We discover and describe limitations which prevent the seamless utilization of existing big data analytics tools and software. Specifically, we evaluate the effectiveness of distributed key-value stores for manipulating large data sets across tightly coupled parallel supercomputers. Although existing distributed key-value stores have proven highly effective in cloud environments, we find their performance on HPC clusters to be degraded. Accordingly, we have built an HPC specific key-value stored called the Multi-Dimensional Hierarchical Indexing Middleware (MDHIM). Using standard big data benchmarks we find that MDHIM performance more than triples that of Cassandra on HPC systems.

1 Introduction

Big data workloads in cloud systems comes from a myriad and diverse set of sources such as sensor networks, the social web, business applications, and data logging. In HPC, large data more typically is created as a result of parallel simulations running on the world's largest supercomputers. The size of data from a single application run can be on the order of petabytes or more in raw files. To be effectively analyzed, these simulation results need to be indexed in multiple dimensions with multiple orderings to find patterns and reveal scientific insight from the data. Indexing and sorting data of these quantities using existing tools may be prohibitively slow or may even be impossible. In addition to these scientific simulations, we have also discovered big data limitations within existing filesystems and storage middleware.

For example, we have found that PLFS (Parallel Log Structured File System) [3], a system software developed at Los Alamos National Laboratory for improving the speed of parallel writes to shared files can generate excessive amounts of metadata for very large files [9]. The limitation is due to how PLFS loads all global metadata for a particular file into every readers' memory. A distributed key-value store (KVS) could be used to distribute this metadata and allow efficient processing of very large PLFS files.

Fortunately, distributed key-value stores exist and have been the subject of a very large amount of excellent research [7, 5, 8, 10]. Unfortunately, these existing tools have typically been designed for cloud environments and are not well-suited for HPC supercomputers and HPC workloads.

Typically, these key-value stores work as follows. They store partitioned data across machines by means of a hashing algorithm that maps keys to specific servers. The hashing algorithms are consistent, meaning that the same keys are always mapped to the same server. Servers and clients communicate through a TCP/IP network, which could be local or geographically distributed. Servers are system daemons configured *a priori* to serve a portion of the key space and may provide failover in the case of a network or system error.

In an HPC environment however, TCP/IP networks may or may not be available, depending on the system. Instead, many HPC systems have specialized interconnects such as Infiniband or proprietary networks from Cray and IBM that provide extremely low latency and very high bandwidth that exceeds TCP/IP capabilities. These specialized interconnects have APIs that support RDMA and/or messaging natively. HPC interconnects often additionally support TCP/IP, however, the performance is lower than the the native APIs.

An additional difficulty is that system services such as persistent daemons are difficult to configure on HPC systems where workflows and job scheduling is managed by a resource manager such as SLURM [17] or MOAB [13]. To set up a system service on a system managed by a resource manager, the user must run the system service immediately before the job runs and it must be run as the user. While this is not impossible to achieve, in practice it can be very difficult to set up correctly. Furthermore, the existing cloud key-value software packages are not intended to run in this manner and may expect access to resources not always available to a standard user.

To address these challenges and limitations requires a new distributed key-value store system designed specifically for HPC supercomputers. Such a system must work with Infiniband and its variants, take advantage of the massive computational resources on the compute system, and run as a library within the application. Accordingly, we have developed the Multi-Dimensional Hashing Indexing Middleware (MDHIM); a parallel key-value store framework designed for HPC systems. MDHIM differs from other key-value stores by supporting HPC networks natively, using dynamic servers that start and end with the application, working with pluggable database backends, ordering a global key space by range instead of hash, using cursor type operations, exposing multiple dimensions, and minimizing network traffic with bulk operations. Each of these will be explained in the following sections.

To demonstrate the performance of MDHIM, we have compared its performance to Cassandra [10] using the Yahoo! Cloud Serving Benchmark [6]. The rest of this paper is organized as follows. Section 2 describes the background, Section 3 explains our design goals, Section 4 details our implementation, Section 5 provides an evaluation of performance. In Section 6, we discuss future work and we conclude in Section 7.

2 Background

There were two main motivations for this work. The first was to address metadata limitations in the existing PLFS implementation. As a log-structured file system, PLFS has metadata describing every write to a file. Exascale data volumes for a single HPC checkpoint file are expected to reach tens of petabytes [4]. Using log-structure in PLFS to store data has been shown to improve parallel bandwidth significantly. However, for large files written by tens of thousands of cooperative processes, the total amount of PLFS metadata can exceed multiple gigabytes [9]. Since PLFS runs on the compute nodes, this is problematic for HPC systems in which memory is scarce. MDHIM solves this problem by distributing the metadata across the compute cluster thereby dramatically reducing PLFS' memory consumption.

A second motivation, and the focus of this paper, is to provide big-data capabilities within an HPC environment for big-data analytics. Many current HPC workflows have pipelined simulation and analysis phases in which the simulation data produces data subsequently consumed by the analysis program [2, 16]. The data sets are typically coordinate data of values within a multi-dimensional physics space. Although these applications currently use POSIX as the staging group between consumer and producer, a key-value store could be used where the coordinate address is the key. In addition to improving access rates for small reads and writes, this approach allows a more natural and semantically meaningful shared access method for the producer and the consumer.

The recent DOE-funded FastForward IO project led by Intel [1] has shown a demand from HPC scientists for these semantic data interfaces to replace the flat-file view long dictated by the POSIX interface. MDHIM is a natural persistent data store for these massive coordinate data sets. The key-value interface allows each data block to be stored as a value with the coordinate location of that block serving as the key.

MDHIM was therefore motivated both to address the PLFS metadata challenge as well as to be a scalable storage mechanism for scientific applications. For scientific applications and PLFS, the data, and the keys to address them, is distributed across the running processes. Some subset of the processes in the application are designated as MDHIM server. These servers partition the key-value data into ranged partitions balanced across the cluster.

3 Implementation

MDHIM is implemented as a library using MPI [15] for communications that can be linked to an application to provide a scalable key-value store. It is embeddable, supports single point to point insertion and retrievals, strong consistency, bulk insertion and retrievals, cursor type operations, pluggable data stores, multiple dimensions, and options for adjusting the distribution of the keys.

When the application calls `mdhimInit`, the range servers are started as threads spawned from the main processes. Each range server is responsible for a certain portion of the key space. When the application wants to get or put key-value pairs, it calls into the MDHIM library which then performs client operations to the previously spawned servers. When the application calls `mdhimClose`, the threads are shutdown. This design makes it easy to use MDHIM in a scientific application.

Keys are currently limited to one hundred bytes and values can be of any size that is allowable by the underlying datastore. Keys types are limited to predefined types: integer, long integer, double, long double, string, and the generic byte for sorting purposes. Values are treated as binary blobs, regardless of the contents. Key-value pairs

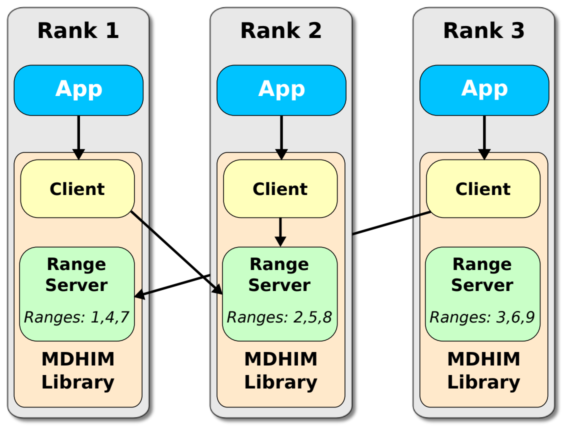

Figure 1: MDHIM's software architecture. Each MPI rank is running an application that is linked to the MDHIM library. The MDHIM library consists of a client and an a range server. In this example, every rank is running a range server. Each MDHIM client is running from the same thread as the application thread that made a call to the MDHIM API. Each range server is running on a separate thread from the client. Every client can contact every range server in response to queries from the application for each target range.

can be inserted and retrieved individually, or in bulk messages. `mdhimPut` and `mdhimGet` will insert or retrieve a single key-value pair. `mdhimBPut` and `mdhimBGet` provide support for bulk messages; up to five hundred thousand key-value pairs can be insertedor retrieved at once. Bulk messages outperform single point to point messages because the number of round-trips on the high speed interconnect is reduced.

Key value pairs can also be retrieved by the API call, `mdhimBGetOp`, which emulates a cursor operation found in many databases. With this function, the user can request to retrieve a number of keys in order starting from a specified key, the first key, or the last key. The direction of the keys retrieved can be ascending or descending. In order for clients to know which range servers to query for these cursor operations, clients must first call the `mdhimStatFlush function`. This function returns statistics from each range server (i.e., min, max, number of keys) that clients use to determine the servers to query. By off-loading query decision making to the clients, we further reduce network traffic and the load on the servers.

Strong consistency is the consistency model used. This was chosen due to the limitations of threads in MPI.

The client portion of the MDHIM library is run from the same thread as the calling application. Clients can contact any range server, even range servers running in the same process. Every key is mapped to a single range server according to the range configuration parameters provided in the `mdhimInit` Figure 1 shows the design of the MDHIM software.

The range server configuration is optionally provided by the application as an initialization parameter that specifies how many range servers should be spawned as well as the size of their ranges. For example, a configuration can specify that every job process can spawn a server or that only some subset of them do so. The number of range servers can affect performance since fewer servers could mean that more clients are communicating with the same servers at the same time and too many servers can result in overly small ranges and more network activity. Ideally, the data should be distributed such that each client is accessing different servers at the same time. In practice, that is difficult to achieve perfectly, but can be tuned using the configurable parameters. For scientific computations whose behavior is mostly consistent across time, a balanced configuration can be found and reused.

MDHIM provides the ability for the user to define how the data is distributed across the range servers with the range size configuration option. The entire key space is a series of ranges that are addressable between the numbers 1 to $2^{32}-1$ and each range can contain at most 2^{64} keys. The size of the range dictates which range servers are responsible for which keys and each key maps to a single range by means of the partitioner. The partitioner makes a decision about which range server handles a key by consulting the mapping algorithms implemented for the different key types mentioned above. These mappings may not be optimal for every workload, however; they provide a reasonable default mapping. Our future work includes support for a user defined partitioner.

Data stores are made pluggable by abstracting the data store interface. New data stores can be added by writing C code that implements the abstracted functions. At this time, support has been added for LevelDB [11] and MySQL [14]. LevelDB is the default data store due to its server-less design and key ordering support. The ordering is necessary to support the cursor operations. MDHIM provides the communication necessary between range servers, thus communication between data store daemons is redundant. However, not all data stores support this design and may still provide good performance depending on the workload. For example, if multiple dimensions are used, some data stores could be optimized for that type of workload.

Multi-dimensionality has been implemented by allowing the user to create any number of secondary index instances. At this time, each index instance creates a new data store instance, if it hasn't already been created, which separates the indexes and allows them to operate independently. Data stores could be implemented to create an index local to the database instead of a new database. Secondary keys can be inserted at the same time as the primary key, or anytime after the primary key has been inserted.

4 Evaluation

To test the performance of MDHIM, we compared it against Cassandra [10] using the Yahoo! Cloud Serving Benchmark [6]. YCSB is a Java based tool for collecting performance results from distributed databases that includes support for many different key-value stores. To use YCSB with MDHIM, we developed an MDHIM plugin for YCSB [12]. The tests were performed on the Mustang cluster at Los Alamos National Laboratory. Mustang consists of 1600 nodes with dual-socket AMD 12-core MagnyCours processors and 64GB of memory per node. The high speed network on Mustang is an Infiniband 4x-QDR interconnect with Mellanox InfiniScale IV hardware configured with a fat tree topology.

The YCSB tests consisted of strong and weak scaling tests to demonstrate the scaling characteristics of both MDHIM and Cassandra. The weak scaling tests performed inserts and reads for 1K and 100K records per node. The strong scaling tests performed insert and read operations on 1 million and 100 million records in total, where the number of keys inserted and read per node reduced as the number of nodes increased. Both types of tests used a uniform key distribution, 1KB per key, one process per node, stored their data on a `tmpfs` mount, and used node sizes ranging from sixteen to two-hundred and fifty-six. The YCSB uniform distribution consists of a specific number or random integers. In all of our tests, the number of random integers available was 1574000. Cassandra was configured to use fifty megabytes for its internal memory tables, to sync its commit log in batch mode, and to use IP over Infiniband. The reason for these changes was to more accurately reflect an HPC workload. Fifty megabytes for memory tables was chosen to reduce the amount of memory used by the server since applications run on the same node as the server process/thread and would need to use most of the memory for computation. The commit log was synced with the batch option instead of periodic since batch mode will require the commit log to be synced to the disk before a response is sent back to the client and MDHIM behaves in the same way. Periodic mode still has to sync the commit log to disk, but can do so in the background while acknowledging client requests immediately. Cassandra cannot take advantage of Inifiniband natively, so it was configured to use Infiniband over IP. MDHIM was configured to use fifty megabytes for memory tables and to use Infiniband natively. In addition, Cassandra was configured to use compression, while MDHIM was not. We found that Cassandra's compression improved it's performance and did not change the database sizes by significant amounts. The results of the Cassandra and MDHIM tests using YCSB are shown in Figures 2 and 3.

The results show that MDHIM outperformed Cassandra in each test. The reasons for MDHIM's increased performance are due to its use of native Infiniband support and a better key distribution. Figure 4 shows the data distribution and sizes from a single test where 128 nodes were used to insert/retrieve 100K records per node for Cassandra and MDHIM. These results show that MDHIM has a more uniform data distribution, which means that all servers served about the same about of data. The sizes of the databases stored by Cassandra and MDHIM are different due to the implementations of Cassandra's internal database and LevelDB. Cassandra's distribution is less uniform, thus leading to an unbalanced server load.

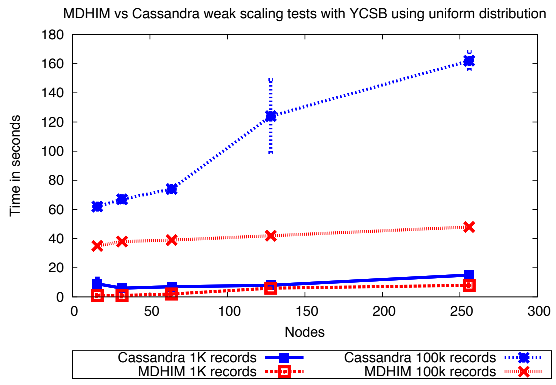

Figure 2: YCSB weak scaling test results for Cassandra and MDHIM. Two types of tests are represented: 1K records per node and 100K records per node. There were three runs performed at each data point and the error bars represent the standard deviation.

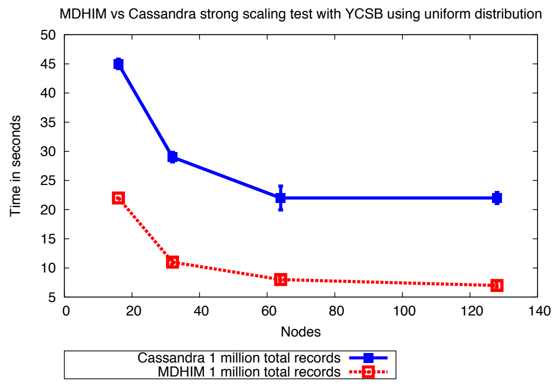

Figure 3: YCSB strong scaling test results for Cassandra and MDHIM. 1 million records were inserted/retrieved in total for each run. There were three runs performed at each data point and the error bars represent the standard deviation.

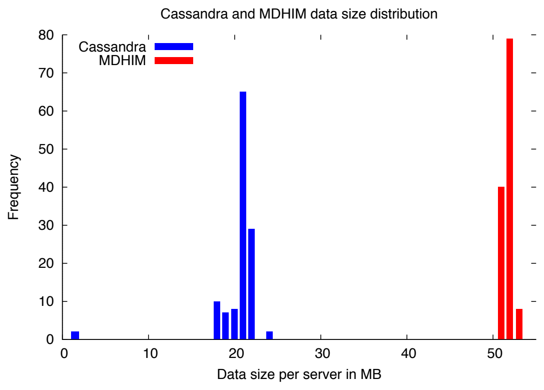

Figure 4: The data size frequency stored per server by Cassandra and MDHIM after a single run with 128 nodes and 100K records inserted per node.

5 Conclusion

In conclusion, MDHIM is a distributed key-value store framework designed for HPC systems. The prototype has been written in MPI to take advantage of high-speed interconnects natively and to be easily embeddable into a scientific application. It is highly scalable and supports multiple dimensions, pluggable data stores, cursor type operations, and bulk messages. Our results with the Yahoo! Cloud Serving Benchmark (YCSB) show that it outperforms Cassandra using a uniform distribution due to it's native high-speed interconnect support and a better key distribution.

6 Acknowledgments

We would like to acknowledge the following individuals for their helpful discussions and feedback: David Bonnie, Aaron Caldwell, Chuck Cranor, Noah Evans, Garth Gibson, Latchesar Ionkov, Jeff Kuehn, Michael Lang, James Nunez, Stephen Poole, Aaron Torres, and Alfred Torrez.

References

[1] Barton, E., Bent, J., and Koziol, Q. Fast forward storage and io program documents, Jun 2006.

[2] Bent, J., Faibish, S., Ahrens, J., Grider, G., Patchett, J., Tzelnic, P., and Woodring, J. Jitter-free co-processing on a prototype exascale storage stack. In *28th IEEE Symposium on Massive Storage Systems and Technologies* (2012), MSST.

[3] Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., and Wingate, M. Plfs: A checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (New York, NY, USA, 2009), SC '09, ACM, pp. 21:1–21:12.

[4] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R. S., Yelick, K., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Keckler, S., Klein, D., Kogge, P., Williams, R. S., and Yelick, K. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead, 2008.

[5] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[6] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[7] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[8] Fitzpatrick, B. Distributed caching with memcached. *Linux J. 2004*, 124 (Aug. 2004), 5–.

[9] He, J., Bent, J., Torres, A., Grider, G., Gibson, G., Maltzahn, C., and Sun, X.-H. I/o acceleration with pattern detection. In *ACM Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, June 2013), HPDC 13.

[10] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[11] Leveldb. google/leveldb, January 2015. `https://github.com/google/leveldb/`.

[12] MDHIM. MDHIM, January 2015. `https://github.com/mdhim/`.

[13] MOAB. Adaptive Computing website, June 2014. `http://www.adaptivecomputing.com`.

[14] MySQL. MySQL :: The world's most popular open source database, January 2015. `http://www.mysql.com/`.

[15] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. *MPI: The Complete Reference*, 2nd ed., vol. 1, The MPI Core. The MIT Press, Cambridge, Massachusetts, 1998.

[16] Thain, D., Bent, J., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Livny, M. Pipeline and Batch Sharing in Grid Workloads. In *Proceedings of High-Performance Distributed Computing (HPDC-12)* (Seattle, Washington, June 2003).

[17] Yoo, A., Jette, M., and Grondona, M. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., vol. 2862 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 44–60.

Notes

[1]LANL publication: LA-UR-15-21957