15-618 Project
K-V Storage for Parallel Architecture

Xinyu Zhang (xinyuzh3), Zhi Lin (zlin2)

May 6, 2022

Abstract. k-v storage has been widely used in algorithms and applications to convert time complexity to space complexity. While many implementations of k-v storage are deployed on the same node, few of them could be used by parallel algorithms and architectures. In this project, we designed and implemented a k-v storage system running on multiple nodes. We set up a master-worker model with a two-level hashing mechanism to define the architecture, and use a modified optimistic asynchronous message passing model to pass the requests and data. Benchmark shows that our system has up to 4.5x parallel speedup for highly parallel k-v data operations. We also show the real application of our system in parallel computing by running a parallel BSGS algorithm.

Keywords: k-v Storage, Two-level Hashing, MPI, Optimistic Message Passing

Project Completeness: We have finished all the items mentioned in our project proposal, including the 125% goal.

Presentation Video Link: `https://youtu.be/rRbvQhK749k`.

1 Introduction

k-v storage is a kind of data structures which store values and objects using a key. It supports search, put and remove operations with low time complexity. K-V storage has been widely used in applications, and most programming languages have implemented K-V storage in the internal library. For example, the map and unordered_map in C++, and Hashmap in Java.

However, as algorithms on the single node show performance bottlenecks, parallel computing architectures are attracting people's attentions now. While many algorithms can split tasks to different nodes, old k-v storage architectures have some problems. (1) The storage space on the same machine is limited, so we cannot store the data in the hash table on a single node. (2) Parallel accessing of data cause race conditions, and locks greatly reduce the performance.

These defects motivate us to create a parallel K-V storage. Based on the issues above, a parallel K-V storage should (1) store data on different nodes. (2) distribute data operations to different nodes. In this project, we (1) designed and implemented an architecture for parallel K-V storage, including a master-worker role model and a modified asynchronous message passing model, (2) designed benchmark tests based on traces to test our system, and (3) tested our system with a real algorithm using hash table: BSGS algorithm. Specifically, we made the BSGS algorithm parallel and generated real traces to feed our k-v storage system.

2 Related Work

The simplest K-V storage is a hash table on a single machine. Most modern languages have implemented this kind of K-V storage. For example, C++ included unordered_map in the Standard Template Library (STL) [1]. It handles hash collisions by forming a linked list or a red-black tree. It does not support multi-threaded operations since they may create rings in the buckets.

To support concurrent operations, application could use mutex and let only one thread access the data structure at the same time. This will harm the performance hugely since it makes parallel operations sequential. An internal data structure called ConcurrentHashMap in Java supports concurrent operations by using fine-grained lock [2]. However, The limitation of storage size on a single node still restrict the application of it in parallel algorithms.

Some big companies developed hash table distributed on different nodes. For example, Microsoft developed FASTER in 2018, which used a concurrent log-structured store and a concurrent hash index to support fast updates [3]. These kind of hash tables involve interactions with disks, so they are more suitable for data centers, and are not very portable for normal parallel algorithms.

As the hash table grows, there may be more and more collisions, so the real searching complexity is much greater than $\mathcal{O}(1)$. To maintain the search performance, data movement is needed. In the implementations in C++ and Java, a variable called load factor will control the movement. Specifically, if the number of data exceeds the load factor multiplies the capacity, the table should be moved to another larger space. Usually, the space will double each time a movement happens, which makes operations which trigger a data movement have a longer latency.

An alternative way is linear hashing [4]. Instead of doing a large scale movement at once, linear hashing performs bucket split and bucket merge during each insert and remove operation. That means, the complexity for each operation will distribute more uniformly, and utilization of space will increase.

3 Design

The basic idea of this project is to support parallel algorithms. Our design is mainly based on this point. Parallel algorithms need to return the result fast, so the data should be stored in the memory instead of the disk.

3.1 Roles Model

We designed two types of nodes in this project: Masters and Workers (Slaves). A master is a node responsible for routing and distributing requests. A worker is where data locates and operations are performed. In practice, there may be another type of nodes called callers. However, we did not include this role in this project, since it does not depend on the KV storage system. There may be multiple master nodes in the system. Each master node are connected to all the worker nodes.

An alternative model is excluding the difference of nodes. For example, any node could receive requests, and forward the requests that should be processed by other nodes. This model is less efficient because synchronization overhead will interact with hash table operation overhead in other nodes. A worse case is dead lock (although with low probability), since each node may wait for a batch of asynchronous requests to be received by other nodes (explained below).

Letting the caller directly send the message to the worker node may be more efficient. However, it is important to make the storage system transparent to users. We don't want users to change the code according to the real worker numbers.

3.2 Two-level Hashing

Since the data is stored on the worker nodes with only one replica, each master should have a deterministic algorithm to decide which node a key belongs to. Besides, we want to improve load balance across all the worker nodes. If a worker node reaches its storage quota, some requests may fail except the system execute data movement or a fast scale-up. Based on these considerations, we implemented a two-level hashing in this project. Specifically, the master will execute a very simple hashing function, where the input is the key, and the output is the worker id.

In this project, we use modular hashing, with the modulus equal to the number of worker nodes. By allocating the number of workers a multiple of 2, we could take advantage of bitwise operations by simply fetching the last several bits of the key. The first level hashing is very simple, so it does not affect the performance of the system even with the fact that the number of masters are usually much smaller than the number of workers.

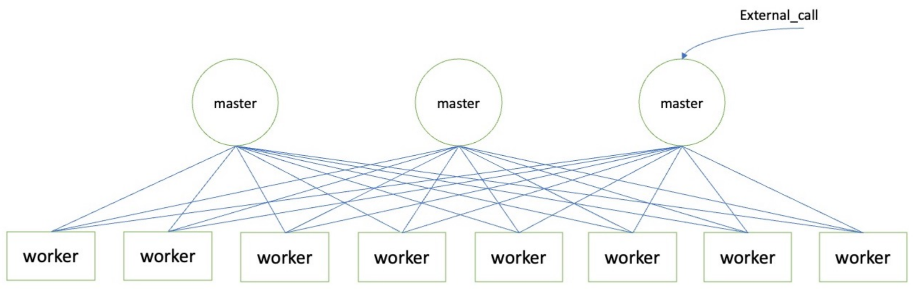

Figure 1: Roles and Architecture

3.3 Protocol

The nodes communicate with each other using MPI. In this project, we applied an optimistic asynchronous message passing model between masters and workers. Specifically, in the code part, the masters will call MPI_Isend() function to send the messages asynchronously, and the workers will call MPI_Recv() function to receive the messages synchronously. The worker does not call asynchronous receive function because worker process is much more complex than master process, so asynchronous receive does not make a difference.

Optimistic asynchronous message passing model has good performance, but it requires storage in the message layer. In order not to exhaust the space on the message layer, at the master nodes, we will call MPI_Wait() after sending a batch of requests. The batch size is set to 100 in the benchmark. The communication protocol could be described by figure 2.

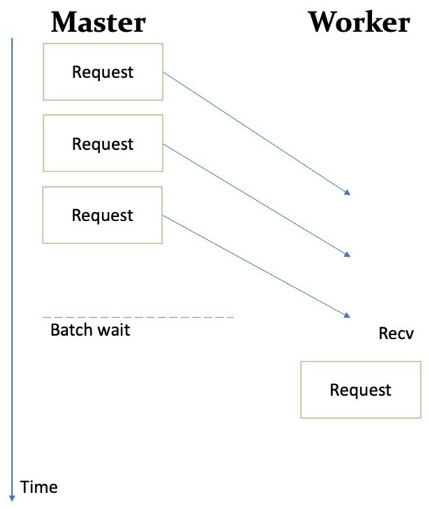

Figure 2: Optimistic MPI Model

For the message content, the message passed from masters to workers include sources, commands (put, remove, get, etc.), key, and value. The message passed from workers to masters include sources and operation results.

We also set up a model with the existence of caller nodes. In this model, after receiving a request from a caller, the master node will immediately return the worker node id, and then forward the request to that worker. The caller will then receive the message directly from that specified worker, as seen in figure 3. This model could eliminate a layer of forwarding of response, and reduce the load of master nodes.

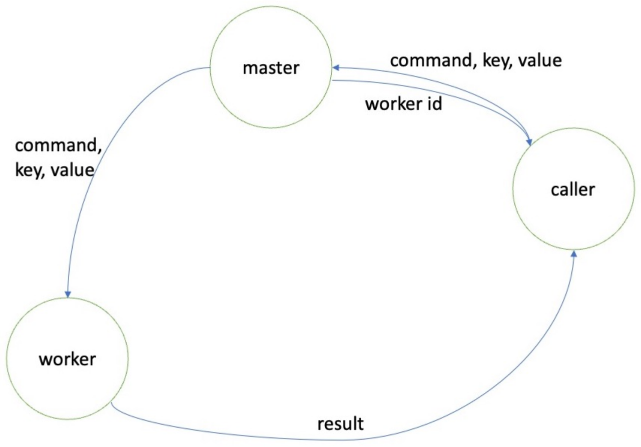

Figure 3: Message model with caller nodes

4 Benchmark and Analysis

4.1 Trace Pattern and Benchmark Strategies

We care mainly about the put and get performance of our system, so we involved these two kinds of operations in our trace files. To simulate the real case, we use a dynamic probability that a trace be 'GET' in our trace generation program. Specifically, at first, all the operations should be 'PUT' since our table is empty. The probability that a 'GET' appears will increase as the number of trace lines goes up. At last, there will always be 'GET' operations. We generated trace files with 100k trace lines, where each line represents a 'PUT' or 'Get' operation.

We wrote Python code to test our system automatically on PSC machine with 128 nodes in shared mode, and integrate the data collected. In our project, we implemented two kinds of underlying hashing functions. The first is linear hashing, and the second is flat hashing. Our test program will test both of them.

4.2 1 Worker to Process Concurrent Requests

In the first test, we would like to test how concurrent requests will affect the performance of a single-node system. We will fix the number of workers as 1, and add the number of masters. As mentioned before, all the masters will read their own trace files, and send requests to the only worker, so generally the workload is growing linearly. The benchmark result could be shown :

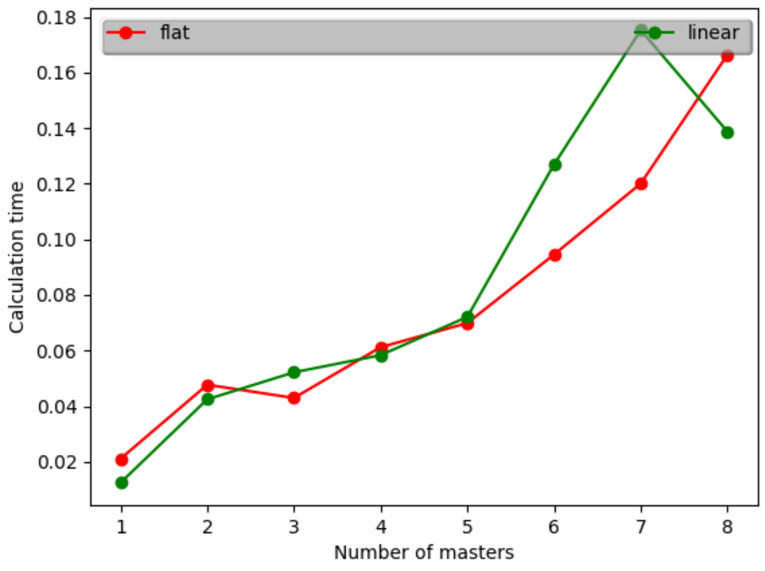

Figure 4: Computation Time with Different Number of Masters

We can see that compared to having only one master, having multiple masters will make the running time much longer. In detail, the running time will grow faster than the master number (super-linear). That's because different masters will interfere with each other. For example, a master will wait for synchronization while the node is executing the requests from another master. This result reveals that K-V storage with single node implementation is not good for parallel algorithms and architectures even without considering the limitation of storage space.

4.3 Fix master and Change Worker

In the second test, we would like to test the performance improvement by using multiple nodes. We will fix the number of masters at 1-8, and then change the number of workers for each master number. The number of workers is similar to the 'thread number' in parallel computing. The results of master number of 2,4,8 are shown in figure 5, 6, and 7. Note that 1 master is equal to sequential situation, so there's no need to use a parallel k-v storage system.

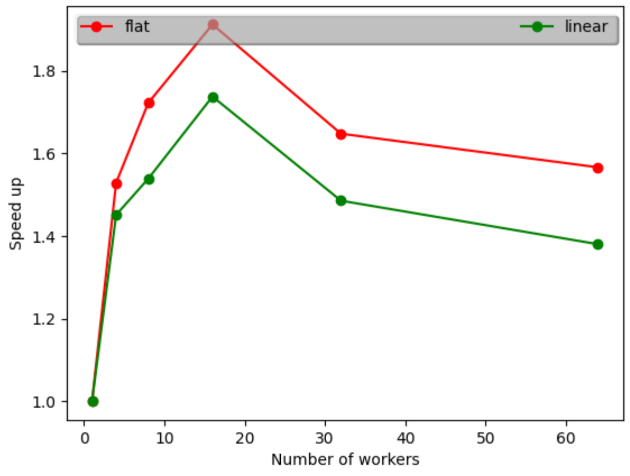

Figure 5: 2 Master

From the results we learn that, adding the number of workers could improve efficiency. However, too many workers might harm the performance (the speedup will decrease) because the synchronization overhead is higher.

Adding the number of masters (adding the level of parallelism) could increase the speedup. Specifically, with 8 masters, we got a maximum speedup of 4.5x at 32 workers. The speedup is smaller than the ideal number because of synchronization overhead and communication overhead through MPI. Flat hashing could gain better speedup with larger number of masters, but a bit worse than linear hashing if the number of masters is small.

The benchmark result also shows that if the ratio of masters and workers is between 1 : 4 to 1 : 8, the speedup could be the best. This could be a direction if our system is applied to the real pratice.

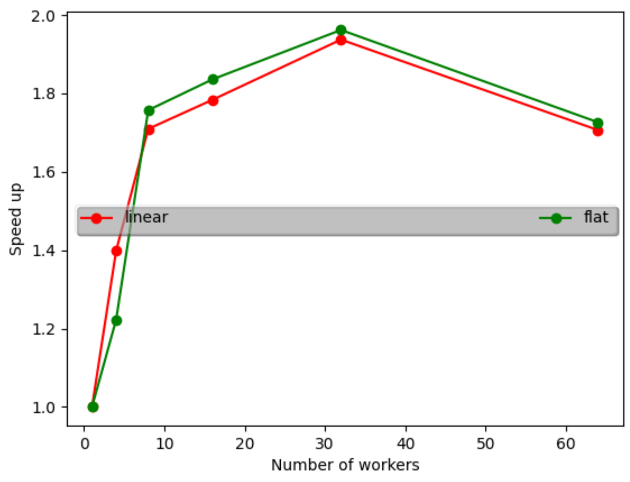

Figure 6: 4 Master

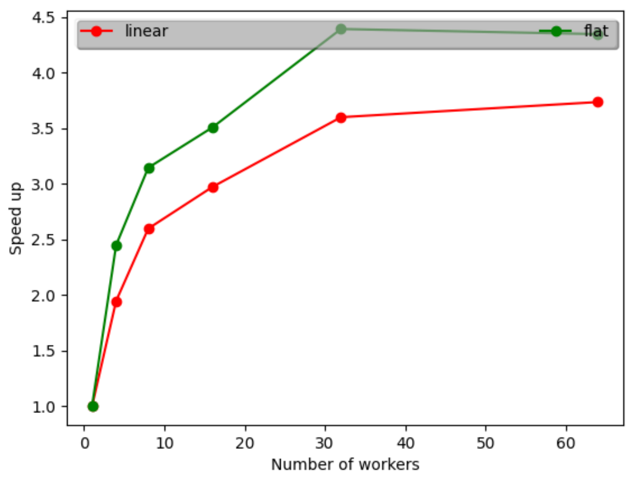

Figure 7: 8 Master

4.4 BSGS fix master 8

The third test is about testing our system in a real algorithm. A classic algorithm using hash table is Shank's Algorithm, also known as "Baby-Step-Giant-Step" (BSGS) algorithm [5]. As we know, discrete logarithm is a computational hard problem in cryptography, so it is a base for public key cryptography, and parallel computer architectures have shown some threat to the security of some old cryptography systems.

A discrete logarithm could be denoted by:

$$A^x = B \mod C \tag{1}$$

The brute force algorithm has a $\mathcal{O}(n)$ complexity, where n is the ordinal of C. BSGS algorithm rewrites the equation as:

$$A^{im+j} = B \mod C \tag{2}$$

where $m = \lceil \sqrt{C-1} \rceil$ if C is a prime number. This equation could also be written as:

$$A^j = A^{-im}B \mod C \tag{3}$$

Therefore, we can first traverse all the A^j and stored them in the hash table, then traverse all the $A^{-im}B$ and check whether one of them exists in the hash table. The overall complexity is $\mathcal{O}(\sqrt{n})$.

BSGS algorithm can be paralleled, since each put and search are independent. Specifically, we can split the put and search operations into k task sets, where k is the (worker) node number. The start point of each worker could be calculated by fast power modulation algorithm. However, the data stored in the hash table by each node should be visible to all other nodes. Our hash table in this project could support parallel BSGS algorithm.

In this test, since we care more about the performance of parallel K-V storage, we translated the BSGS algorithms to trace files and test them on our system instead of starting callers running the BSGS algorithm interactively. This is reasonable because the only kind of time costing operations in the BSGS algorithm are hash table operations. We ran BSGS algorithm with 1 master 1 worker (serial) and 8 master 64 worker (parallel). The result is shown in table 1:

Table 1: Running time and Parallel Speedup of BSGS Algorithm

Serial (1 Master 1 Worker)	Parallel (8 Master, 64 Worker)	Speedup
0.0663278	0.00635374	10.4391

We observed super-linear speedup in this parallel algorithm. That's because the speedup should be divided into two parts: multiple masters (runners of the algorithm) speedup and multiple workers speedup. Since multiple masters speedup should not exceed $x8$ in most situations, the result clearly reveals that our system could support parallel algorithms well compared to single-node solutions.

5 Conclusion

In this project we designed and implemented a k-v storage system for parallel computing. The benchmark results show that our system could support parallel algorithms using kv storage very well. We also give directions for the number of masters and workers in practice.

There are still some limitations of our work. For example, we did not take the communication overhead between the callers and the masters into consideration. In practice, the master could not read requests from trace files directly. It should receive the request from the caller where the parallel algorithms are really running on. Besides, our system needs to set the number of masters and workers at the start of the program. However, the load might change while running a parallel program. Maintaining a constant number of masters and workers may waste some calculation resources. Ideally, we could do some dynamic scaling with scale-out and scale-in of both master nodes and worker nodes.

1. References

[1] "std::unordered_map," Available at https://www.cplusplus.com/reference/unordered_map/unordered_map/ Accessed: 2022-05-04.

[2] "Class concurrenthashmap<k,v>," Available at https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html Accessed:2022-05-04.

[3] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: A concurrent key-value store with in-place updates," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 275–290.

[4] W. Litwin, "Linear hashing: a new tool for file and table addressing." in *VLDB*, vol. 80, 1980, pp. 1–3.

[5] D. Shanks, "Class number, a theory of factorization, and genera," in *Proc. of Symp. Math. Soc., 1971*, vol. 20, 1971, pp. 41–440.