Hybrid Parallel Data Structure: Balancing Strong and Local Linearizability

https://github.com/levleyfer/parallelProgrammingproject.git

By Daniel Fraimovich 315880963
Lev Leyfer 321385064

Abstract

Concurrency in distributed systems often needs a delicate trade-off between performance and consistency. Linearizability, which is the usual correctness rule for concurrent objects, makes sure operations look like they happen instantly and follow real-time order. But strict linearizability can slow performance a lot. Two well-known alternatives are **strong linearizability** and **local linearizability**:

Strong Linearizability tries to keep a stricter order of operations than usual linearizability. This can provide better guarantees for some applications, but it might add cost.

Local Linearizability lets different parts of the system handle their own operations in a more relaxed way, which might give better performance but can weaken the global consistency guarantees.

This paper summarizes and synthesizes two academic works:

1. "Strong Linearizability using Primitives with Consensus Number 2" (Attiya, Castaneda, Enea, 2024)
2. "Local Linearizability" (Haas et al., 2015)

Both works explore ways to enhance performance while maintaining acceptable levels of consistency. In this paper we will propose a hybrid parallel data structure that selectively applies strong and local linearizability, aiming to optimize both correctness and efficiency.

Summary of Articles

Strong Linearizability Using Primitives with Consensus Number 2

Focusing on *strong linearizability*, this paper examines whether various concurrent objects can be implemented in a wait-free (or lock-free) strongly-linearizable manner using low-level primitives whose consensus number is 2 (like test&set or fetch&add). The main finding is that, although some objects of consensus number 1 do have wait-free strongly-linearizable implementations derived from such primitives, many objects with consensus number 2 do not. In particular, queues, stacks, and their relaxed variants are shown to lack lock-free strongly-linearizable solutions under these operations. The authors base their conclusions on formal proofs involving impossibility results for consensus, complemented by concrete algorithmic examples that highlight when and why strong linearizability fails with limited synchronization primitives.

Local Linearizability

This paper introduces the concept of *local linearizability*, a weaker consistency condition than traditional linearizability. The central claim is that, by allowing different parts (or components) of a concurrent data structure to be linearized independently, local linearizability reduces the global synchronization overhead and enables more scalable implementations. To support this claim, the authors present both theoretical arguments—showing that local linearizability still preserves key correctness guarantees—and experimental results demonstrating efficient, high-performing implementations of locally linearizable queues and stacks. They show that by splitting the system into parts (using a decomposition principle), each part's ordering can be checked on its own, making the overall approach simpler for system designers to understand and verify.

Connection Between the Articles

Both papers focus on balancing correctness and performance in concurrent data structures. The first paper suggests *loosening* linearizability (local linearizability) to improve real-world performance, while the second shows that *strengthening* it (strong linearizability) can be unworkable with limited synchronization operations.

Despite these different angles, both share the same goal: They help us understand how different consistency models affect both theory and actual performance. They explain a range of linearizability, from local too strong, and suggest ways to build systems that offer strong guarantees but still scale well.

Suggestions: Hybrid Parallel Data Structure

Motivation

The goal is to balance consistency and performance dynamically, depending on the operation type.

Concept

We propose a Hybrid Parallel Data Structure that selectively applies strong linearization for critical operations and local linearization for actions that require local consistency and hybrid operations that combine both of the concepts for adaptable actions.

Key Features

❖ Local Operations: These simulate lightweight, shard-specific actions that only require local consistency (demoLocalOperations).

- ❖ Critical Operations: These represent global actions requiring strong ordering and atomicity across the entire system (demoCriticalOperation).

- ❖ Hybrid Operations: Combines local writes with global updates, showcasing the dynamic nature of hybrid consistency (demoHybridOperation).
  function first writes to a local shard and then increments a global counter. This simulates a scenario where an operation's local state must also be reflected in a globally consistent manner.

Expected Impact

By thoughtfully combining local linearizability's scalability benefits with the correctness of strong linearizability for key operations and the creation of the combined hybrid between the 2 concepts, our new data structure serves as a compelling solution to the challenges outlined in the original papers.

and may improve efficiency in real-world applications like **distributed databases, high-performance computing, and cloud storage**.

- ❖ **Improved Throughput**: Reducing unnecessary global synchronization improves performance.
- ❖ **Flexible Consistency Guarantees**: Allows prioritization of critical transactions while optimizing background processes.

OverView of the Hybrid Data structure

<u>Local Data and Shards- local linearizable:</u>

The data structure maintains a list of dictionaries (shards), each protected by its own lock. This "sharding" helps reduce lock contention when many threads update different parts of the local data.

How Threads Are Assigned:

Depending on the total number of threads, the model creates:

- A few threads that perform only local operations,
- A few that perform only critical (global) updates, and
- A few that perform hybrid operations (both local and global).

<u>Global Counter - Strong linearizable:</u>

there's a single global counter updated under its own lock. This "critical" variable is updated in two ways:

critical_update that simply adds a small increment (here, 1).

hybrid_operation that updates both the local shard (using local_write) and the global counter with a larger, random value.

Local Reads/Writes:

The structure also keeps counters for the number of local reads and writes performed.

Mismatch between the global counter and the local sum.

- The hybrid operations add roughly the same amount to both.
- The local operations add extra value to the local shards (but not to the global counter).
- The standalone critical operations add a small increment to the global counter.

This mismatch will allow us to evaluate the model and the running operation between each number of running threads 1-64.

Implementation & Evaluation

Project implementation & tests in Python

```
import threading
import time
import random

class HybridDataStructure:
    def __init__(self, num_shards=4):
        self.num_shards = num_shards
        self.local_locks = [threading.Lock() for _ in range(num_shards)]
        self.local_data = [{} for _ in range(num_shards)]
        self.global_lock = threading.Lock()
        self.global_counter = 0
        self.local_reads = 0
        self.local_writes = 0

    def _get_shard_index(self, key):
        return hash(key) % self.num_shards

    def local_write(self, key, value):
        shard_index = self._get_shard_index(key)
        with self.local_locks[shard_index]:
            if key in self.local_data[shard_index]:
                self.local_data[shard_index][key] += value
            else:
```

```python
                self.local_data[shard_index][key] = value
            self.local_writes += 1

    def local_read(self, key):
        shard_index = self._get_shard_index(key)
        with self.local_locks[shard_index]:
            self.local_reads += 1
            return self.local_data[shard_index].get(key, None)

    def critical_update(self, increment=1):
        with self.global_lock:
            self.global_counter += increment

    def hybrid_operation(self, key, value):
        self.local_write(key, value)
        self.critical_update(increment=value)

    def print_stats(self):
        print(f"Total Reads: {self.local_reads}, Total Writes:
{self.local_writes}")

def demo_local_operations(hybrid_obj, thread_id):
    for _ in range(5):
        key = f"user_{random.randint(1, 3)}"
        if random.random() < 0.5:
            val = random.randint(100, 999)
            hybrid_obj.local_write(key, val)
        else:
            hybrid_obj.local_read(key)
        time.sleep(random.random() * 0.1)

def demo_critical_operation(hybrid_obj, thread_id):
    for _ in range(5):
        hybrid_obj.critical_update(increment=1)
        time.sleep(random.random() * 0.1)

def demo_hybrid_operation(hybrid_obj, thread_id):
    for _ in range(5):
        key = f"resource_{thread_id}"
        val = random.randint(1000, 2000)
        hybrid_obj.hybrid_operation(key, val)
        time.sleep(random.random() * 0.1)

def run_test(num_threads, hybrid_obj):
    threads = []

    for i in range(num_threads // 3):
        t = threading.Thread(target=demo_local_operations,
args=(hybrid_obj, i))
        threads.append(t)

    for i in range(num_threads // 3):
        t = threading.Thread(target=demo_critical_operation,
args=(hybrid_obj, i))
        threads.append(t)

    for i in range(num_threads // 3):
        t = threading.Thread(target=demo_hybrid_operation,
args=(hybrid_obj, i))
        threads.append(t)

    while len(threads) < num_threads:
        t = threading.Thread(target=demo_hybrid_operation,
args=(hybrid_obj, len(threads)))
        threads.append(t)

    start_time = time.time()
    for t in threads:
        t.start()

    for t in threads:
        t.join()
    end_time = time.time()

    total_local_sum = sum(sum(shard.values()) for shard in
hybrid_obj.local_data)
    difference = hybrid_obj.global_counter - total_local_sum

    print(f"Test with {num_threads} threads completed in {end_time -
start_time:.4f} seconds")
```

```python
    print(f"Final Global Counter = {hybrid_obj.global_counter}")
    print(f"Total Local Sum = {total_local_sum}")
    print(f"Difference = {difference}")
    print("Final Local Data:")
    for index, shard in enumerate(hybrid_obj.local_data):
        print(f"Shard {index}: {shard}")
    hybrid_obj.print_stats()
    print("-" * 40)

if __name__ == "__main__":
    thread_counts = [1, 2, 4, 8, 16, 32, 64]
    for count in thread_counts:
        hybrid_ds = HybridDataStructure(num_shards=4)
        run_test(count, hybrid_ds)
```

Project implementation & tests in Java (for Running in Moshe's Server)

The test for Hybrid Parallel Data Structure

```java
import java.util.*;

import java.util.concurrent.*;

class MppRunner {

    private static final Random random = new Random();


    private static void demoLocalOperations(MppRunnerCode hybridObj) {

        for (int i = 0; i < 5; i++) {

            String key = "user_" + (random.nextInt(3) + 1);

            if (random.nextBoolean()) {

                int val = random.nextInt(900) + 100;

                hybridObj.localWrite(key, val);

            } else {

                hybridObj.localRead(key);

            }
            try { Thread.sleep((long) (random.nextDouble() * 100)); }
catch (InterruptedException ignored) {}

        }

    }


    private static void demoCriticalOperation(MppRunnerCode hybridObj) {

        for (int i = 0; i < 5; i++) {

            hybridObj.criticalUpdate(1);

            try { Thread.sleep((long) (random.nextDouble() * 100)); }
catch (InterruptedException ignored) {}

        }

    }


    private static void demoHybridOperation(MppRunnerCode hybridObj, int
threadId) {

        for (int i = 0; i < 5; i++) {

            String key = "resource_" + threadId;

            int val = random.nextInt(1001) + 1000;

            hybridObj.hybridOperation(key, val);

            try { Thread.sleep((long) (random.nextDouble() * 100)); }
catch (InterruptedException ignored) {}

        }

    }


    public static void runTest(int numThreads, MppRunnerCode hybridObj) {

        List<Thread> threads = new ArrayList<>();

        int partitionSize = numThreads / 3;
```

```java
        for (int i = 0; i < partitionSize; i++) {
            final int index = i; // Capture the index for each thread
            threads.add(new Thread(() -> demoLocalOperations(hybridObj)));
        }

        for (int i = 0; i < partitionSize; i++) {
            final int index = i; // Capture the index for each thread
            threads.add(new Thread(() ->
demoCriticalOperation(hybridObj)));
        }

        for (int i = 0; i < partitionSize; i++) {
            final int index = i; // Capture the index for each thread
            threads.add(new Thread(() -> demoHybridOperation(hybridObj,
index)));
        }

        while (threads.size() < numThreads) {
            final int index = threads.size(); // Use the current thread
size as the index
            threads.add(new Thread(() -> demoHybridOperation(hybridObj,
index)));
        }

        long startTime = System.currentTimeMillis();
        threads.forEach(Thread::start);
        threads.forEach(t -> {
            try { t.join(); } catch (InterruptedException ignored) {}
        });
        long endTime = System.currentTimeMillis();


        int totalLocalSum = hybridObj.getTotalLocalSum();
        int difference = hybridObj.getGlobalCounter() - totalLocalSum;


        System.out.println("Test with " + numThreads + " threads completed
in " + (endTime - startTime) + " ms");
        System.out.println("Final Global Counter = " +
hybridObj.getGlobalCounter());
        System.out.println("Total Local Sum = " + totalLocalSum);
        System.out.println("Difference = " + difference);
        hybridObj.printStats();
        System.out.println("----------------------------------------");
    }


    public static void main(String[] args) {
        int[] threadCounts = {1, 2, 4, 8, 16, 32, 64};
        for (int count : threadCounts) {
            MppRunnerCode hybridObj = new MppRunnerCode(4);  // Fixed
instantiation
            runTest(count, hybridObj);
        }
    }
}
```

The code Hybrid Parallel Data Structure

```
import java.util.*;

import java.util.concurrent.*;

import java.util.concurrent.atomic.AtomicInteger;

import java.util.concurrent.locks.*;


class MppRunnerCode {

    private final int numShards;

    private final Lock[] localLocks;

    private final Map<String, Integer>[] localData;

    private final Lock globalLock;

    private final AtomicInteger globalCounter;

    private int localReads = 0;

    private int localWrites = 0;


    public MppRunnerCode(int numShards) {

        this.numShards = numShards;

        this.localLocks = new ReentrantLock[numShards];

        this.localData = new ConcurrentHashMap[numShards];

        for (int i = 0; i < numShards; i++) {

            this.localLocks[i] = new ReentrantLock();

            this.localData[i] = new ConcurrentHashMap<>();

        }

        this.globalLock = new ReentrantLock();

        this.globalCounter = new AtomicInteger(0);

    }
    private int getShardIndex(String key) {

        return Math.abs(key.hashCode()) % numShards;

    }


    public void localWrite(String key, int value) {

        int shardIndex = getShardIndex(key);

        localLocks[shardIndex].lock();

        try {

            localData[shardIndex].merge(key, value, Integer::sum);

            localWrites++;

        } finally {

            localLocks[shardIndex].unlock();

        }

    }


    public Integer localRead(String key) {

        int shardIndex = getShardIndex(key);

        localLocks[shardIndex].lock();

        try {

            localReads++;

            return localData[shardIndex].getOrDefault(key, null);

        } finally {

            localLocks[shardIndex].unlock();

        }

    }


    public void criticalUpdate(int increment) {
```

```java
        globalLock.lock();

        try {

            globalCounter.addAndGet(increment);

        } finally {

            globalLock.unlock();

        }

    }


    public void hybridOperation(String key, int value) {

        localWrite(key, value);

        criticalUpdate(value);

    }


    public void printStats() {

        System.out.println("Total Reads: " + localReads + ", Total Writes: 
" + localWrites);

    }


    public int getGlobalCounter() {

        return globalCounter.get();

    }


    public int getTotalLocalSum() {

        return Arrays.stream(localData).mapToInt(shard -> 
shard.values().stream().mapToInt(Integer::intValue).sum())

                .sum();

    }

}
```

Performance Evaluation

We will evaluate the **Hybrid Parallel Data Structure** based on:

Execution Time

How long does it take for different numbers of threads?

Does performance scale efficiently

Correctness (Final Global Counter Validation)

Ensures all increments were performed correctly (no race conditions).

Results

Running Locally in python:

```
Test with 1 threads completed in 0.3685 seconds

Final Global Counter = 8631

Total Local Sum = 8631

Difference = 0

Final Local Data:

Shard 0: {}

Shard 1: {}

Shard 2: {}

Shard 3: {'resource_0': 8631}

Total Reads: 0, Total Writes: 5

----------------------------------------

Test with 2 threads completed in 0.2653 seconds

Final Global Counter = 13858

Total Local Sum = 13858

Difference = 0

Final Local Data:

Shard 0: {}

Shard 1: {}
Shard 2: {}

Shard 3: {'resource_0': 7674, 'resource_1': 6184}

Total Reads: 0, Total Writes: 10

----------------------------------------

Test with 4 threads completed in 0.3370 seconds

Final Global Counter = 15602

Total Local Sum = 16383

Difference = -781

Final Local Data:

Shard 0: {}

Shard 1: {'user_1': 625, 'user_3': 161}

Shard 2: {'resource_3': 7975}

Shard 3: {'resource_0': 7622}

Total Reads: 3, Total Writes: 12

----------------------------------------

Test with 8 threads completed in 0.3136 seconds

Final Global Counter = 31632

Total Local Sum = 36788

Difference = -5156

Final Local Data:

Shard 0: {}

Shard 1: {'user_3': 3870, 'resource_6': 8101, 'user_1': 678}

Shard 2: {'resource_7': 7906}

Shard 3: {'resource_0': 7436, 'resource_1': 8179, 'user_2': 618}

Total Reads: 2, Total Writes: 28

----------------------------------------

Test with 16 threads completed in 0.3776 seconds

Final Global Counter = 45709

Total Local Sum = 51877
```

```
Difference = -6168

Final Local Data:

Shard 0: {'resource_4': 7282}

Shard 1: {'user_3': 2892, 'resource_2': 7405, 'user_1': 1036}

Shard 2: {'resource_3': 8235}

Shard 3: {'resource_0': 7109, 'resource_1': 7651, 'resource_15': 8002, 'user_2': 2265}

Total Reads: 14, Total Writes: 41

----------------------------------------

Test with 32 threads completed in 0.3594 seconds

Final Global Counter = 89916

Total Local Sum = 107461

Difference = -17545

Final Local Data:

Shard 0: {'resource_4': 6955, 'resource_9': 7441, 'resource_31': 6828}

Shard 1: {'user_3': 5686, 'user_1': 4530, 'resource_2': 7493, 'resource_6': 6315, 'resource_30': 6740}

Shard 2: {'resource_3': 7913, 'resource_5': 8636, 'resource_7': 9102}

Shard 3: {'resource_0': 7837, 'resource_1': 7955, 'resource_8': 6651, 'user_2': 7379}

Total Reads: 19, Total Writes: 91

----------------------------------------

Test with 64 threads completed in 0.4430 seconds

Final Global Counter = 164125

Total Local Sum = 188442

Difference = -24317

Final Local Data:

Shard 0: {'resource_4': 6998, 'resource_9': 6916, 'resource_16': 7666}

Shard 1: {'user_3': 9198, 'user_1': 7881, 'resource_2': 6617, 'resource_6': 7801, 'resource_11': 7080,
'resource_13': 8135, 'resource_20': 7756}

Shard 2: {'resource_3': 7537, 'resource_5': 8005, 'resource_7': 7431, 'resource_10': 6366, 'resource_12':
8113, 'resource_18': 7746, 'resource_63': 7387}
```

```
Shard 3: {'user_2': 7343, 'resource_0': 7782, 'resource_1': 7692, 'resource_8': 7529, 'resource_14': 8182,
'resource_15': 7253, 'resource_17': 7256, 'resource_19':

 6772}

Total Reads: 60, Total Writes: 155
```

Results running in server:

```
Test with 1 threads completed in 177 ms

Final Global Counter = 8154

Total Local Sum = 8154

Difference = 0

Total Reads: 0, Total Writes: 5

----------------------------------------

Test with 2 threads completed in 241 ms

Final Global Counter = 15061

Total Local Sum = 15061

Difference = 0

Total Reads: 0, Total Writes: 10

----------------------------------------

Test with 4 threads completed in 313 ms

Final Global Counter = 14900

Total Local Sum = 16019

Difference = -1119

Total Reads: 3, Total Writes: 12

----------------------------------------

Test with 8 threads completed in 320 ms

Final Global Counter = 31239

Total Local Sum = 35087

Difference = -3848
```

```
Total Reads: 3, Total Writes: 27

----------------------------------------

Test with 16 threads completed in 363 ms

Final Global Counter = 41861

Total Local Sum = 51207

Difference = -9346

Total Reads: 9, Total Writes: 46

----------------------------------------

Test with 32 threads completed in 447 ms

Final Global Counter = 85818

Total Local Sum = 97539

Difference = -11721

Total Reads: 26, Total Writes: 84

----------------------------------------

Test with 64 threads completed in 386 ms

Final Global Counter = 171430

Total Local Sum = 196268

Difference = -24838

Total Reads: 54, Total Writes: 161

----------------------------------------
```

The results demonstrate how the hybrid model balances consistency and performance under increasing thread counts.

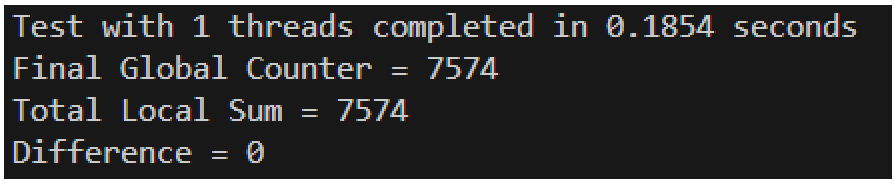
we will evaluate each run with the following observations:

- Execution Time
- Global Counter - **Strong linearizable updates**
- Total Local Sum - **Local linearizable writes**
- Consistency Gap - **Trade-off between speed and consistency**
- Shard Distribution - **Hash collisions** of **Local linearizability hashmap**

<u>1 thread test</u>

Only hybrid operations run. Therefore, the local sum (which is the sum of values in the shards) matches exactly the global counter.

With no contention, the global lock and shard locks operate optimally. All updates are atomic and consistent.

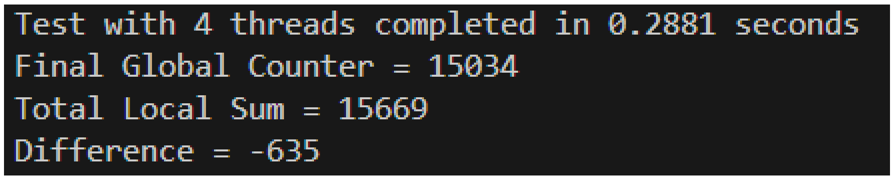


```
Test with 1 threads completed in 0.1854 seconds
Final Global Counter = 7574
Total Local Sum = 7574
Difference = 0
```

<u>4 thread run</u>

With a mix of operations, you see a small difference (–635), meaning that the extra local updates (from the local-only operations) increased the local sum relative to the global counter, which only got a +1 per critical operation.

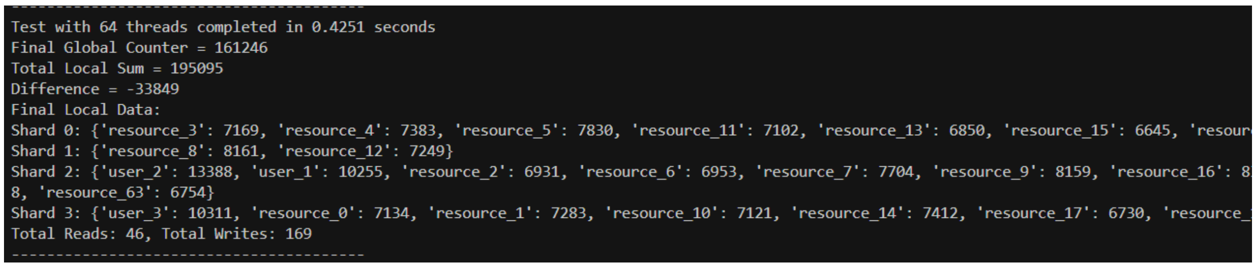
Threads writing to the same shard experience lock contention. We also see there is starting to become a consistency gap, that we measure by subtracting (Total Local Sum - Final Glovbal Counter).
The single global lock delays critical updates, causing partial desynchronization.



```
Test with 4 threads completed in 0.2881 seconds
Final Global Counter = 15034
Total Local Sum = 15669
Difference = -635
```

<u>64 Threads run</u>

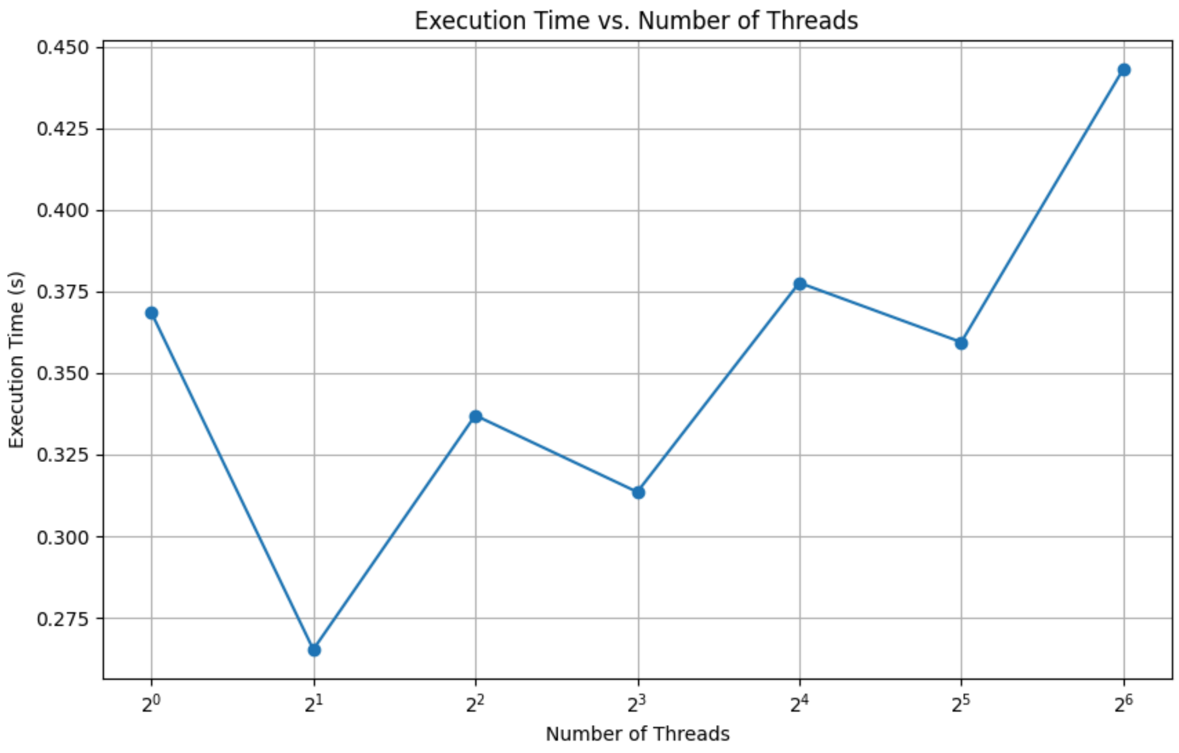
As thread count increases, the total values (global counter and local sum) increase along with the number of operations performed. However, the discrepancy (difference) grows (in absolute value), showing the imbalance between operations that update one counter vs. the other.



```
----------------------------------------
Test with 64 threads completed in 0.4251 seconds
Final Global Counter = 161246
Total Local Sum = 195095
Difference = -33849
Final Local Data:
Shard 0: {'resource_3': 7169, 'resource_4': 7383, 'resource_5': 7830, 'resource_11': 7102, 'resource_13': 6850, 'resource_15': 6645, 'resour
Shard 1: {'resource_8': 8161, 'resource_12': 7249}
Shard 2: {'user_2': 13388, 'user_1': 10255, 'resource_2': 6931, 'resource_6': 6953, 'resource_7': 7704, 'resource_9': 8159, 'resource_16': 8
8, 'resource_63': 6754}
Shard 3: {'user_3': 10311, 'resource_0': 7134, 'resource_1': 7283, 'resource_10': 7121, 'resource_14': 7412, 'resource_17': 6730, 'resource_
Total Reads: 46, Total Writes: 169
----------------------------------------
```

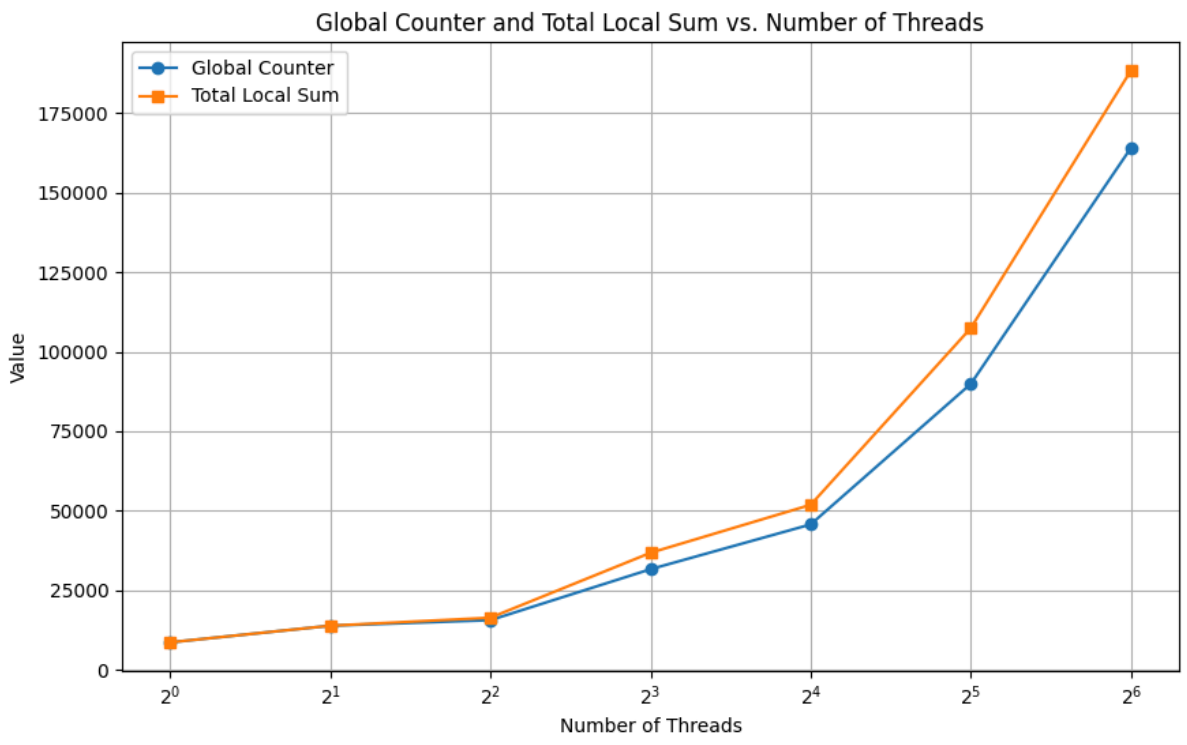
Analysis

This plot shows the relationship between execution time and the number of threads in your multi-threaded hybrid data structure.



At 32 and 64 threads, the execution time increases. This may be due to **lock contention**, where threads are competing for access to shared resources (global counter or shard locks). As more threads are added, they spend more time waiting for locks, slowing down overall performance.

This plot shows the relationship between the **number of threads** and the values of the **Global Counter** and **Total Local Sum**.



At higher thread counts (32 and 64), the gap between the Total Local Sum and the Global Counter widens significantly. This suggests potential **data inconsistency**, likely caused by the following:

Concurrency Issues: Some writes may be missed due to contention or thread synchronization issues.**Lock Contention:** Threads may not be writing to the local shards or updating the global counter in a perfectly synchronized way.

Better Consistency at Lower Thread Counts
With fewer threads (1, 2, 4), the values are closer together, indicating better consistency and less contention.

As the number of threads increases, contention and synchronization issues cause the Global Counter to lag behind the Total Local Sum. This suggests potential improvements in locking mechanisms or synchronization strategies to ensure better consistency.

Conclusion & Future Directions

This paper presented a hybrid approach to concurrent data structures, integrating **strong and local linearizability** to optimize both correctness and efficiency by selectively applying different concepts.

The results and plots illustrate both the benefits and challenges of using a hybrid data structure:

Benefits:

- Sharding reduces contention, allowing many threads to update local data concurrently.
- Hybrid operations maintain a balance between local and global state when used in isolation.

Challenges:

- When mixing different types of operations, the consistency between the global counter and the local shard sums can diverge.
- Care must be taken in the design if the application requires these values to remain in sync.

This experiment serves as a practical demonstration of concurrency trade-offs and is a useful starting point for discussions on designing scalable, thread-safe data structures.

References

1. **Attiya, H., Castaneda, A., & Enea, C.** (2024). *Strong Linearizability using Primitives with Consensus Number 2*. Technion, UNAM, Ecole Polytechnique.
2. **Haas, A., Henzinger, T. A., Holzer, A., et al.** (2015). *Local Linearizability*. TU Wien, University of Salzburg, Google Inc., Meta, University of Cambridge.