Concurrent Data Structure: dequeue

Yujia Zhu (yz10317), Chengying Wang(cw4450)

Abstract

In the multi-core processor-dominated era of digital computing, enhancing concurrent control strategies is critical for optimizing multi-threaded applications. This necessity is particularly pronounced when managing complex algorithms such as graph algorithms and certain iterative machine learning algorithms, where conventional methods like lock-free mechanisms, lock-based systems, and node replication face issues including complexity, performance instability, and high computational costs.

This study introduces a fine-grained locking strategy specifically designed for concurrent deques. The technique involves employing locks at both ends of the deque to support efficient parallel operations and substantially reduce the risk of lock contention and deadlocks.

The findings reveal that despite some overhead, the fine-grained locking approach significantly enhances performance by optimizing the frequency of lock operations. This project not only addresses existing gaps in high-concurrency scenarios but also provides a scalable and widely applicable solution that improves the efficiency and stability of concurrent data structures.

1 Introduction

In today's rapidly digitizing world, the ubiquity of multi-core processors has made concurrent programming increasingly important. Efficient concurrent control strategies are crucial for enhancing program performance and response times, especially when processing algorithms such as Breadth-First Search (BFS). However, existing concurrent control strategies, including lock-free mechanisms, lock-based mechanisms and node replication techniques, still face various limitations in high-concurrency scenarios. For instance, although lock-free mechanisms can theoretically improve performance, they are complex to implement and may become unstable under uneven access patterns; lock-based mechanisms, while straightforward and intuitive, can introduce performance bottlenecks and deadlock risks; node replication techniques, while improving performance in multi-node systems, are limited by their high spatial occupancy and potential blocking nature.

To address these issues, this study proposes a fine-grained locking strategy for implementing concurrent deques to enhance the efficiency and scalability of concurrent control. This strategy employs locks at both ends of the deque to support efficient parallel operations in a multi-threaded environment, significantly improving the data structure's throughput and response speed while reducing the risk of lock contention and blocking.

Moreover, Breadth-First Search (BFS) is just one example among many potential applications of concurrent deques that require efficient concurrent access. Similar applications include real-time computing systems, task scheduling systems, and high-performance network applications, all of which need rapid and frequent updates to and accesses of data structures to optimize processor resource usage or efficiently handle large numbers of concurrent requests. The fine-grained locking strategy proposed in this project not only addresses the deficiencies of existing concurrent control strategies in specific high-concurrency application scenarios but can also be extended to other complex data structures for concurrent operations, further improving the efficiency and stability of concurrent operations.

Through the analyses provided, this research not only responds to the limitations of existing concurrent control strategies in specific high-concurrency application scenarios but also offers a solution with broad applicability. This helps to advance the application and development of concurrent data structures more widely, meeting the demands of modern multi-threaded applications for high-performance concurrent control.

2 Literature Survey

2.1 Classification of Concurrent Control Strategies

1**. Lock-Free Mechanisms**

Relevant Work: Lock-free mechanisms employ atomic operations, such as Compare-And-Swap (CAS), to synchronize between threads. This approach is commonly used in high-performance computing scenarios, such as processor-intensive concurrent applications.

Advantages:

- **Performance Improvement**: By reducing lock contention, lock-free mechanisms can enhance the system's parallelism and performance [1].
- **Avoidance of Blocking**: Lock-free data structures eliminate blocking issues, reduce system latency, and simplify concurrency management [3].

Disadvantages:

- **Implementation Complexity**: Lock-free mechanisms are typically more complex to implement than lock-based mechanisms, requiring developers to possess advanced concurrent programming skills [1].
- **Performance Instability**: In highly competitive environments, performance may degrade due to thread data dependencies and memory access delays [1].

2. **Lock-Based Mechanisms**

Relevant Work: Lock-based mechanisms manage access to shared resources by introducing mutex locks (e.g., mutexes, read-write locks), which are the most traditional and widespread concurrent control strategies.

Advantages:

- **Simplicity**: Locks are straightforward and intuitive tools for concurrency control, easy to implement and understand [3].
- **Data Consistency**: Locks effectively prevent data races, ensuring the atomicity and orderliness of operations [3].

Disadvantages:

- **Performance Bottlenecks**: In high-concurrency environments, lock contention may significantly reduce performance [3].
- **Deadlock Risk**: Improper management of locks may lead to deadlocks, increasing system complexity [3].

3. **Node Replication Technique**

Relevant Work: The node replication technique, especially on NUMA architectures, reduces the need for cross-node data synchronization by replicating data structures on each node.

Advantages:

- **Performance and Scalability**: Enhances performance in multi-node systems, especially in environments with high operation conflicts [2].
- **Universality and Automation**: Automatically converts any single-threaded data structure to a concurrent structure, reducing development difficulty [2].

Disadvantages:

- **High Space Overhead**: Replicating entire data structures to each processing node requires more memory [2].
- **Blocking Nature**: Certain operations may lead to overall process blocking, especially in scenarios with frequent updates [2].

2.2 Limitations of Existing Work and the Necessity of This Project

In exploring implementations of concurrent deques, although existing concurrent control strategies have their merits, they still show certain limitations in high-concurrency scenarios such as Breadth-First Search (BFS). Lock-free mechanisms, while theoretically enhancing performance, are complex and may become unstable in practice, especially under uneven access patterns or intense competition. Lock-based mechanisms, though easy to use, can introduce performance bottlenecks and deadlock issues in high-concurrency settings.

Node replication techniques can effectively boost performance in multi-node systems but may not be suitable for memory-sensitive or high-responsiveness required applications due to their high spatial occupancy and potential blocking nature. Breadth-First Search (BFS) is just one example among the applications of concurrent deques that require efficient concurrent access.

Similar applications include:

1. **Real-Time Computing**: Such as real-time data processing and event-driven systems, which require quick and frequent updates and access to data structures.

2. **Task Scheduling Systems**: In operating systems or high-performance computing, task scheduling systems may need to add or remove tasks concurrently to optimize processor resource usage.

3. **Network Applications**: Such as high-performance network servers using non-blocking I/O models, where data structures need to efficiently handle a large number of concurrent connections and requests.

To address these needs, this project proposes a fine-grained locking strategy that aims to provide a solution that maintains the simplicity and intuitiveness of concurrent control while effectively alleviating performance bottlenecks and deadlock risks in high-concurrency environments. This strategy allows multiple threads to operate in parallel at different ends of the deque, significantly enhancing throughput and response speed while greatly reducing the likelihood of lock contention and blocking. The application of fine-grained locks is not limited to deques but can be extended to other complex data structures like graph structures and indexing structures, further improving the efficiency and stability of concurrent operations. Through the above analysis, this project not only addresses the deficiencies of existing concurrent control strategies in specific high-concurrency application scenarios but also provides a solution with broad applicability, helping to promote the application and development of concurrent data structures in a wider range.

3 Proposed Idea

3.1 Choice of Locking Mechanisms

In the selection of concurrency control strategies, our preference for locking mechanisms over CAS (Compare-And-Swap)-based lock-free implementations is driven by several key considerations:

1. Simplicity and Maintainability: Lock implementations are generally simpler and more intuitive than CAS-based lock-free mechanisms. This reduces development complexity and facilitates future code maintenance and upgrades. Herlihy and Shavit underscore the importance of selecting appropriate lock types for managing performance and system complexity, reinforcing our decision to use straightforward mutex locks[6].

2. Ease of Programming: Locking mechanisms simplify the creation of thread-safe code by automatically managing synchronization and mutual exclusion among threads, whereas lock-free implementations require manual management of state changes, increasing programming complexity.

3. Stability of Performance: Locking mechanisms offer relatively stable performance, suitable for scenarios where concurrency operations are not extremely frequent. Conversely, lock-free implementations can exhibit significant performance variability in high-conflict environments. Additionally, lock algorithms have proven more reliable than lock-free algorithms in managing significant delays during process execution[4].

4. Avoidance of ABA Problems: Locks prevent the ABA problem that can arise with lock-free implementations, enhancing the stability and reliability of data structures.

3.2 Choosing Standard Locks Over Read-Write Locks

Despite the advantages of read-write locks in scenarios that allow multiple concurrent read operations, we have opted for standard locks (mutexes) for the following reasons:

1. System Complexity and Maintenance Costs: The management of read-write locks is relatively complex, involving more intricate logic processing that could lead to additional runtime overhead, especially in low concurrency environments.
2. Runtime Overhead: Standard locks typically incur lower runtime costs than read-write locks, as they do not require additional counters and logic to manage concurrent read and write operations. Bueso's research supports this point, indicating that standard locks are more efficient in multicore systems[5].

3. Memory Overhead: Standard locks require less memory as they do not need to track the number of read operations or manage complex lock states, thereby enhancing system efficiency.
4. Lock Competition and Fairness: Standard locks simplify the mechanism for resolving competition, reducing the potential for write operation starvation and ensuring the fairness of operation serialization.

3.3 Details of Parallel Implementation

Parallelization Strategy:

This project employs OpenMP's locking mechanisms by configuring separate locks at both ends of the deque (`headLock` and `tailLock`), achieving efficient parallel operations. The core advantage of this separate lock strategy lies in allowing different threads to operate independently at the head and the tail of the dequeue, thereby minimizing lock contention and thread blocking.

Implementation Details:

- Initialization and Destruction:
    - Initialization (`initDeque_c`): At initialization, head and tail pointers are set to `NULL`, and locks for both ends are initialized with `omp_init_lock`, ensuring thread safety from the outset.
    - Destruction (`destroyDeque_c`): During destruction, the entire dequeue is traversed to free each node, and `omp_destroy_lock` is called to dismantle the head and tail locks, preventing memory leaks and ensuring robustness of the program.
- Operations:
    - Adding Elements (`push_front_c` and `push_back_c`): When adding elements, locks at the respective ends are engaged using omp_set_lock. New nodes are inserted at either the head or the tail, updating pointers as necessary, followed by releasing the locks with `omp_unset_lock`.
    - Removing Elements (`pop_front_c` and `pop_back_c`): Removal operations start by locking the relevant end. If the queue is not empty, the respective node is removed, pointers are updated, and locks are released after the operation.

Synchronization and Performance Considerations:

- Lock Granularity: The use of fine-grained locking (separate head and tail locks) reduces the likelihood of lock contention, allowing for nearly unhindered simultaneous operations by different threads in high-concurrency scenarios.

- Exception Handling: Exception handling logic is incorporated to ensure locks are properly released in cases such as memory allocation failures or operations on an empty dequeue, preventing deadlocks.

3.4 Rationale and Advantages of Parallelization

The choice to implement a concurrent deque using OpenMP locking mechanisms is intended to meet the performance and maintainability needs of modern multi-threaded applications. This approach enhances parallel processing efficiency and flexibility through several key benefits:

1. Improved Parallel Efficiency: Independent locks at the dequeue head and tail significantly reduce lock contention. This separation allows operations in a multi-threaded environment to be logically independent, enabling threads to operate on different ends without blocking each other, substantially improving concurrent access efficiency to the data structure.

2. Reduced Thread Blocking: By allowing multiple threads to work simultaneously on different parts of the dequeue, the system minimizes thread blocking and waiting times. This separated lock strategy not only enhances response speeds but also optimizes resource allocation on multicore processors, allowing each core to perform tasks more efficiently.

3. Simplified Programming and Debugging: Compared to complex lock-free programming models, using clear lock mechanisms simplifies the logic of concurrency control, making it easier for developers to implement and maintain synchronization among threads. This not only reduces development complexity but also decreases the challenges encountered during debugging in a multi-threaded environment.

Advantages of Parallelization:

1. Operational Flexibility: Independent locks provide high flexibility, enabling the queue to perform parallel operations without interference under high load conditions, such as allowing one thread to frequently add elements at the head while another thread removes elements at the tail—critical for real-time data processing systems.
2. Broad Applicability: This parallelization design maintains high performance even in low-concurrency environments due to reduced lock competition, making the deque suitable for a wide range of applications from high-concurrency real-time processing to low-concurrency data logging.

3. Data Consistency and Integrity: Lock mechanisms ensure atomicity and orderliness of data operations, maintaining the consistency and integrity of the data structure during concurrent modifications—vital for systems reliant on precise data handling, such as financial transaction platforms.

By implementing these parallelization strategies and their associated benefits, this project's concurrent deque is designed to meet high performance, high availability, and ease of management requirements, making it an ideal choice for handling complex data structures in a multi-threaded environment.

4 Experimental Setup

To test the concurrent dequeue implemented with locks, experiments are carried out on crunchy machines on NYU CIMS servers, which supports CPU and memory intensive processes. Detailed information and setup about this environment is as follows:

- CPU - Four AMD Opteron 6272 (2.1 GHz) (64 cores)
- Memory - 256 GB
- OS - CentOS 7

Executable file of our program is compiled with gcc, using the -fopenmp option to support OpenMP speedup on multicore. Detailed program compiling and running specification is attached in the README file.

In this experiment, we compared two implementations of dequeues: a non-concurrent version and a concurrent version that utilizes locks. We tested these implementations under two distinct settings and scenarios, which are detailed below. Prior to testing, we set the number of threads for the concurrent version. We conducted two types of benchmark tests for both the non-concurrent and concurrent versions.

The first category of benchmark tests encompasses a wide series of basic operations on dequeue, such as `push_back`, `push_front`, `pop_back`, and `pop_front`. These tests are designed to evaluate the performance of our concurrent dequeue implementation under various conditions.

The second type of benchmark test case is specifically designed to apply dequeue within a particular scenario: BFS (Breadth-First Search) on a graph. To evaluate the performance of two distinct dequeues, we have intensified CPU utilization by testing BFS on a complete graph with a substantial number of vertices.

We selected BFS on a graph as a benchmark because, in BFS and certain shortest path algorithms, the queue plays an essential role in traversing or enumerating vertices. Consequently, it is crucial to ensure that a concurrent version of the dequeue, which can also function as a queue, operates correctly and efficiently within these graph algorithms.

5 Experiments and Analysis

In this section, we will perform experiments using both the non-concurrent dequeue and its concurrent counterpart on the two types of benchmark programs previously mentioned. Our objective is that our implementation of a concurrent dequeue using OpenMP for parallel programming could outperform its non-concurrent counterpart.

These experiments will primarily assess three metrics: execution time, speedup, and efficiency. Here, "execution time" is measured by using `time` command, and "speedup" is defined as the ratio of the time taken using a single thread to the time taken using 'n' threads. "Efficiency" is calculated as the speedup achieved with 'n' threads divided by the number of threads.

5.1 Basic dequeue operations

In our initial experiment, we evaluated the performance of a deque when accessed by multiple threads performing a high volume of operations. The operations were distributed evenly among four types: `push_back`, `push_front`, `pop_back`, and `pop_front`, with each type accounting for one-quarter of the total operations. We measured the execution time of these basic operations across various thread configurations and problem sizes. The results are presented in Table 1.

Table 1 Execution time(s) of basic dequeue operations testing

Problem size	Number of threads			
	1	2	4	8
N = 1000	0.0050	0.0070	0.0080	0.0100
N = 10000	0.0080	0.0230	0.0340	0.0570
N = 100000	0.0350	0.1980	0.2770	0.5510

The data presented in the table above clearly shows that merely executing basic dequeue operations in parallel does not optimize execution time as initially anticipated. Instead, for a fixed problem size, the execution time actually increases with the addition of more threads. What's more, as the problem size expands, the execution time correspondingly rises.

The execution speed and efficiency of multi-threading are presented in the following Tables 2 and 3. Table 2 shows results similar to those in Table 1, indicating that the speedup decreases as both the number of threads and the problem size increase. In Table 3, the efficiency is notably low, almost approaching zero. This suggests that employing concurrent dequeue exclusively for numerous basic operations primarily prevent race conditions through locking mechanisms. However, this approach does not significantly impact the running time or improve speedup, as it introduces substantial overhead.

Table 2 Speedup of basic dequeue operations testing

Problem size	Number of threads			
	1	2	4	8
N = 1000	1.0000	0.7143	0.6250	0.5000
N = 10000	1.0000	0.3478	0.2353	0.1404
N = 100000	1.0000	0.1768	0.1264	0.0635

Table 3 Efficiency of basic dequeue operations testing

Problem size	Number of threads			
	1	2	4	8
N = 1000	1.0000	0.3571	0.1563	0.0625
N = 10000	1.0000	0.1739	0.0588	0.0175
N = 100000	1.0000	0.0884	0.0316	0.0079

Based on the data from Tables 1 and 2, we observe that the execution time and speedup increase as the number of threads increases, contrary to our expectations. This trend may primarily be attributed to the overhead associated with initiating, terminating, and switching between threads. As the number of threads grows, the cumulative costs of context switching can escalate, potentially negating the advantages of parallel processing.

Additionally, we incorporate locking mechanisms into our concurrent dequeue structure to manage access during dequeue operations. The frequent activation and deactivation of these locks contribute significantly to the degradation of both time efficiency and speedup. This high volume of lock manipulation increases the time cost, undermining the potential gains from parallel processing.

5.2 Graph BFS

In our upcoming experiment, we aim to assess the performance of the dequeue operation when applied to a graph algorithm—specifically, Breadth-First Search (BFS), where dequeue is used as a queue in this algorithm. In each exploration of a node, the adjacent nodes are traversed to judge whether it can be added to the queue for future exploration. Tests are conducted on a complete graph, characterized by edges connecting every pair of vertices. Additionally, we measured the execution time of these basic operations across a range of thread configurations and problem sizes. Detailed execution time is in the following Table 4.

Table 4 Execution time(s) of graph BFS testing using dequeue

Problem size	Number of threads			
	1	2	4	8
N = 1000	0.0130	0.0130	0.0120	0.0140
N = 10000	0.5320	0.3020	0.1830	0.1370
N = 100000	50.4720	25.4770	13.1150	7.0210

From Table 4, it is evident that when the problem size is relatively small, the execution time remains nearly identical across different thread counts. However, as the problem size increases, there is a marked improvement in execution time with the use of more threads. This improvement can be attributed to the distribution of tasks and resources among all threads, effectively reducing the total execution time. Additionally, the strategic use of multiple threads coupled with dequeue locks contributes to enhanced performance. In each iteration of BFS, the program concurrently searches for all possible vertices, adds them to the queue, and then performs a pop_head operation, which involves securing a lock on the head only once. This method optimizes resource use and speeds up the execution process.

The following Tables 5 and 6 show the speedup and efficiency of dequeue when addressing graph BFS with different problem sizes.

Table 5 Speedup of graph BFS testing using dequeue

Problem size	Number of threads			
	1	2	4	8
N = 1000	1.0000	1.0000	1.0833	0.9286
N = 10000	1.0000	1.7616	2.9071	3.8832
N = 100000	1.0000	1.9811	3.8484	7.1887

From Table 5, we can observe that for a small problem size, the speedup remains equal to 1 across all thread counts, implying that there is no gain from parallelism. As the problem size increases, the speedup also increases with more threads. For N = 10000, moving from 1 to 4 threads nearly doubles the speedup, and going to 8 threads more than quadruples it. The largest problem size shows the most significant benefit from threading, with speedup values increasing substantially as more threads are utilized, reaching a speedup of over 7 times with 8 threads.

Table 6 Efficiency of graph BFS testing using dequeue

Problem size	Number of threads			
	1	2	4	8
N = 1000	1.0000	0.5000	0.2708	0.1161
N = 10000	1.0000	0.8808	0.7268	0.4854
N = 100000	1.0000	0.9905	0.9621	0.8986

From Table 6, we can conclude that the efficiency tends to decrease as the number of threads increases, and with larger problem sizes, the efficiency decreases are less dramatic. For the largest problem size, efficiency remains high across all thread counts, only dropping slightly with the increase in threads. This suggests that for large-scale problems, the parallel processing benefits are realized even with the added overhead of threading.

In summary, these tables above suggest that the effectiveness of multi-threading with deques in graph BFS operations depends on the problem size. Larger problems benefit more from parallel processing with multiple threads, while smaller problems may not benefit as much due to the overhead associated with thread management.

6 Conclusions

In this project, the lock mechanism facilitated by OpenMP simplifies the implementation of concurrent dequeue and ensures the prevention of race conditions during dequeue operations. However, this approach does not improve performance. The extensive utilization of various operations leads to frequent activation and deactivation of locks, which can hinder performance rather than enhance it due to the overhead associated with managing these locks.

In specific scenarios, such as graph algorithms or certain iterative machine learning algorithms that process lists or matrices, using a concurrent dequeue with a locking mechanism can ensure correct outcomes and achieve significant speedup with multiple threads. Optimizing the organization of dequeue operations to minimize the frequency of setting and unsetting locks could further enhance performance.

Last but not the least, our graph tests show that concurrent dequeue is crucial in addressing graph-related problems in real-world applications, particularly in those that require high concurrency. For transportation routing or real-time recommendation systems, the ability to efficiently process and update large graphs with many simultaneous operations is essential. To sum up, the use of concurrent deques ensures that these complex, interconnected systems remain robust and efficient, even under the strain of high concurrency demands.

References

[1] A. Rukundo, A. Atalar, and P. Tsigas, "Performance Analysis and Modelling of Concurrent Multi-access Data Structures," in Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'22), July 11–14, 2022.

[2] I. Calciu, S. Sen, M. Balakrishnan, M. K. Aguilera, "How to Implement Any Concurrent Data Structure," in Communications of the ACM, vol. 61, no. 12, pp. 97-103, December 2018.

[3] M. Moir and N. Shavit, "Concurrent Data Structures," Sun Microsystems Laboratories, in Documentation of Concurrent Programming Techniques, pp. 1-23, 2001.

[4] Michael, M. M., & Scott, M. L. (1996). Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing (pp. 267-275).

[5] Davidlohr Bueso. Scalability Techniques for Practical Synchronization Primitives. In: Communications of the ACM, pages 1–9, 2017.

[6] Maurice Herlihy and Nir Shavit. The Art of Multiprocessor Programming. Revised Reprint, Morgan Kaufmann, pages 1–508, 2012.