Synchronization Concurrent Double-Linked List

RAVI SHEKHAR TIWARI (SE23MAID014)

30 September 2023

Assignment Question

Create a concurrent double-linked list (sorted list) using the following synchronization techniques:

- Coarse-grain Synchronization
- Fine-grain Synchronization
- Optimistic Synchronization
- Lazy Synchronization
- Non-blocking Synchronization

Verify the performance of the concurrent data structure for different problem sizes **(2×10^3, 2×10^4, and 10^5)** by varying the number of threads **(1, 2, 4, 6, 8, 10, 12, 14, and 16)** and workloads **(0C-0I-50D,50C-25I-25D, and 100C-0I-0D)**. Consider an average of five trials and the duration of each trial is 10 seconds. Finally, draw appropriate plots using the GNU plot

1 Coarse-grain Synchronization

Coarse-grain synchronization with respect to a doubly linked list involves protecting the entire data structure (the doubly linked list) with a lock or synchronization mechanism. In this context, I'll provide an in-depth explanation of how coarse-grain synchronization is applied to a doubly linked list.

1.1 Doubly Linked List Overview

A doubly linked list is a data structure consisting of nodes, where each node contains two references: one pointing to the previous node and one pointing to the next node. It allows for efficient insertions and deletions at both the beginning and end of the list and enables traversal in both directions.

1.2 Coarse-Grain Synchronization for a Doubly Linked List

- **Single Lock for the Entire List**: In a coarse-grain synchronization approach, the entire doubly linked list is protected by a single lock. This means that any operation (insertion, deletion, or traversal) that needs to access the list must first acquire this lock.
- **Lock Acquisition and Release**: When a thread wants to perform an operation on the doubly linked list, it must first attempt to acquire the lock. If another thread already holds the lock, the requesting thread will be blocked and must wait until the lock becomes available. Once the lock is acquired, the thread can safely perform its operation.

1.3 Benefits

- **Simplicity**: Coarse-grain synchronization with a single lock simplifies the implementation of operations on the linked list. Developers do not need to manage locks at the node level, making the code easier to understand and maintain.
- **Reduced Contention**: Since only one thread can access the list at a time, there is no contention for individual nodes. This can reduce the complexity of managing concurrent access.

1.4 Drawbacks

- **Limited Concurrency**: Coarse-grain synchronization can lead to limited concurrency, especially if multiple threads frequently need to access different parts of the list simultaneously. Threads may spend a significant amount of time waiting for the lock to be released, leading to under utilization of available resources. Performance Bottlenecks: If certain operations

on the doubly linked list take a long time to complete, holding the lock for the entire list can create performance bottlenecks, as it prevents other threads from making progress.

Coarse-grain synchronization in the context of a doubly linked list problem involves using a single lock to protect access to the entire list. Let's break down how this synchronization technique might perform under different problem sizes, thread counts, and workloads:

1.5 Problem Sizes (2×10^3, 10^4, and 10^5)

As the problem size increases (i.e., the number of elements in the doubly linked list), the potential for contention among threads also increases, especially if multiple threads need to access the list simultaneously. Larger problem sizes may result in longer execution times for individual operations (e.g., insertion, deletion) on the list.

1.6 Thread Counts (1, 2, 4, 6, 8, 10, 12, 14, and 16)

With coarse-grain synchronization and a single lock, the performance is likely to be affected by the number of threads contending for access to the list. As the number of threads increases, the contention for the lock also increases. This can lead to situations where threads spend significant time waiting to acquire the lock, reducing overall concurrency and potentially causing performance bottlenecks.

1.7 Workloads (0C-0I-50D, 50C-25I-25D, and 100C-0I-0D)

Different workloads represent different patterns of operations on the doubly linked list: C (read or traversal), I (insertion), and D (deletion). In cases where the workload is predominantly read-heavy (e.g., 100C-0I-0D), coarse-grain synchronization might perform reasonably well because read operations do not require exclusive access to the list and can be performed concurrently. However, for workloads with significant insertions or deletions, the single lock protecting the entire list can become a significant bottleneck, especially if multiple threads attempt to modify the list simultaneously (e.g., 50C-25I-25D). In summary, coarse-grain synchronization with a single lock for a doubly linked list can lead to performance limitations, particularly when dealing with larger problem sizes, higher thread counts, and workloads that involve frequent insertions or deletions. Threads may contend for the lock, leading to reduced concurrency and potential performance bottlenecks.

To mitigate these issues, one might consider alternative synchronization strategies, such as fine-grain synchronization (protecting individual nodes or smaller portions of the list with separate locks), lock-free algorithms, or non-blocking data structures. These alternatives can allow for better parallelism and reduced contention in scenarios with diverse workloads and high thread counts.

However, they are typically more complex to implement than coarse-grain synchronization. The choice of synchronization technique should be based on a careful analysis of the specific requirements and characteristics of the problem at hand.

1.8 Results

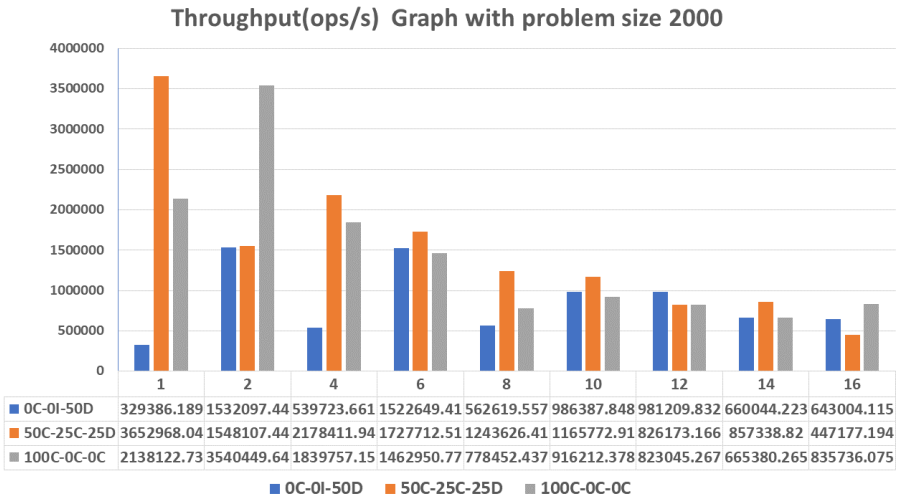


Figure 1: Coarse-Grain Throughput of 2000 problem size on different threads and workload

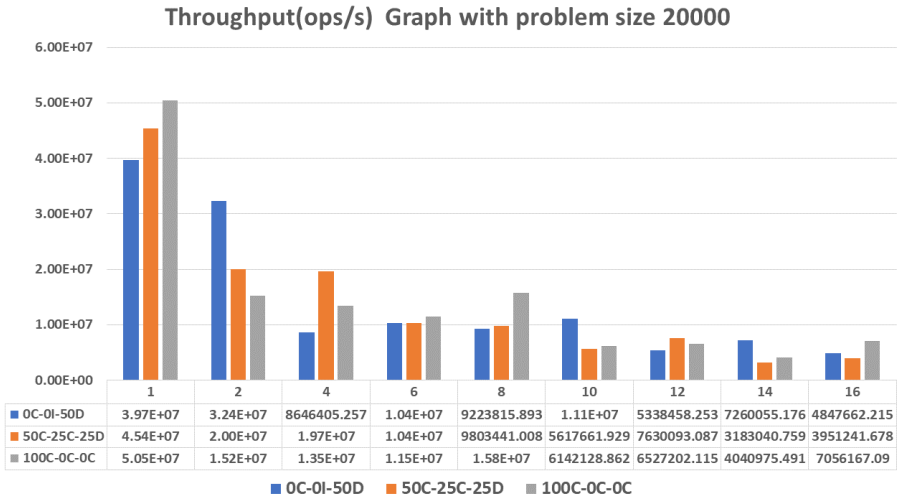


Figure 2: Coarse-Grain Throughput of 20000 problem size on different threads and workload

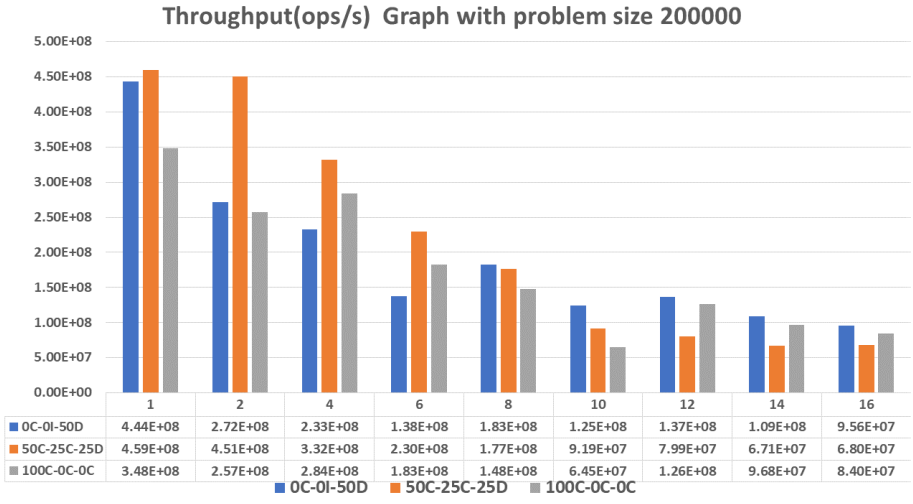


Figure 3: Coarse-Grain Throughput of 200000 problem size on different threads and workload

2 Fine-grain Synchronization

Fine-grain synchronization, in the context of a doubly linked list, involves protecting smaller, more granular portions of the list, such as individual nodes or groups of nodes, with separate locks or synchronization mechanisms. This ap-

proach allows for greater concurrency among threads, as multiple threads can access different parts of the list simultaneously without blocking each other. Let's explore fine-grain synchronization in depth with respect to a doubly linked list:

2.1 Fine-Grain Synchronization for a Doubly Linked List

- **Locks at the Node Level**: In fine-grain synchronization, instead of using a single lock to protect the entire list, locks are associated with each node or a group of nodes in the doubly linked list. These locks are used to control access to specific parts of the list, allowing multiple threads to operate concurrently on different sections.
- **Granularity Control**: The level of granularity in fine-grain synchronization can vary. You can choose to lock individual nodes or group nodes into segments and protect each segment with a separate lock. The granularity should be selected based on the specific access patterns and workload of the application.

2.2 Benefits

- **Improved Concurrency**: Fine-grain synchronization allows multiple threads to access different parts of the list simultaneously, reducing contention and improving parallelism. Threads can perform insertions, deletions, or traversals in parallel without waiting for a global lock.
- **Reduced Locking Overhead**: Fine-grain locks limit the scope of contention. Threads only need to acquire locks for the specific nodes they intend to modify or access, reducing the chances of lock contention and lock-related overhead.
- **Scalability**: Fine-grain synchronization can lead to better scalability in multi-core or multi-processor systems because it minimizes the chances of threads blocking each other.

Fine-grain synchronization in the context of a doubly linked list, when applied to different problem sizes, thread counts, and workloads, can have significant implications for the performance and scalability of concurrent applications. Let's explore how fine-grain synchronization behaves under these varying conditions:

2.3 Problem Sizes ($\mathbf{2 \times 10^3}$, $\mathbf{2 \times 10^4}$, and 10^5)

- **Smaller Problem Sizes**: When dealing with smaller problem sizes, fine-grain synchronization might not provide a substantial performance advantage because there may be fewer nodes to operate on concurrently. However, it can still help in scenarios where threads need to perform frequent updates to the list, as it can reduce contention for locks.

- **Larger Problem Sizes**: As the problem size increases, fine-grain synchronization becomes more critical. With many nodes in the doubly linked list, the likelihood of concurrent access by multiple threads also increases. Fine-grain locks allow threads to operate on different parts of the list concurrently, potentially leading to better performance and scalability.

2.4 Thread Counts (1, 2, 4, 6, 8, 10, 12, 14, and 16)

- **Low Thread Counts**: When there are only a few threads, fine-grain synchronization may not provide a significant advantage over coarse-grain synchronization. Threads may not contend heavily for access to the list, and the overhead of managing fine-grain locks could outweigh the benefits.
- **High Thread Counts**: Fine-grain synchronization shines when there are many threads contending for access to the list, especially in write-heavy workloads (insertions and deletions). With fine-grain locks, threads can work on different parts of the list simultaneously, reducing contention and allowing for better parallelism.

2.5 Workloads (0C-0I-50D, 50C-25I-25D, and 100C-0I-0D)

- **Read-Heavy Workloads (0C-0I-50D)**: Fine-grain synchronization may not provide significant benefits in this workload because read operations do not require exclusive access to the list. Threads can read nodes concurrently without fine-grain locking.
- **Mixed Workloads (50C-25I-25D)**: Fine-grain synchronization can be beneficial in scenarios with mixed workloads. Threads performing inserts and deletes can access different parts of the list concurrently, reducing contention and improving performance. Concurrent read operations can also proceed without blocking.
- **Write-Heavy Workloads (100C-0I-0D)**: Fine-grain synchronization is especially useful in write-heavy workloads where many threads need to insert or delete nodes. Without fine-grain locks, these threads would heavily contend for access, leading to bottlenecks. Fine-grain locks enable high parallelism in such scenarios.

In summary, fine-grain synchronization for a doubly linked list is most beneficial in scenarios with larger problem sizes, high thread counts, and workloads that involve frequent insertions and deletions. It enables threads to work concurrently on different parts of the list, reducing contention and improving parallelism. However, it also comes with increased complexity in implementation and potential overhead, so it should be carefully considered based on the specific requirements of the application. The choice of synchronization strategy should balance the need for concurrency with the associated complexity and overhead.

2.6 Results

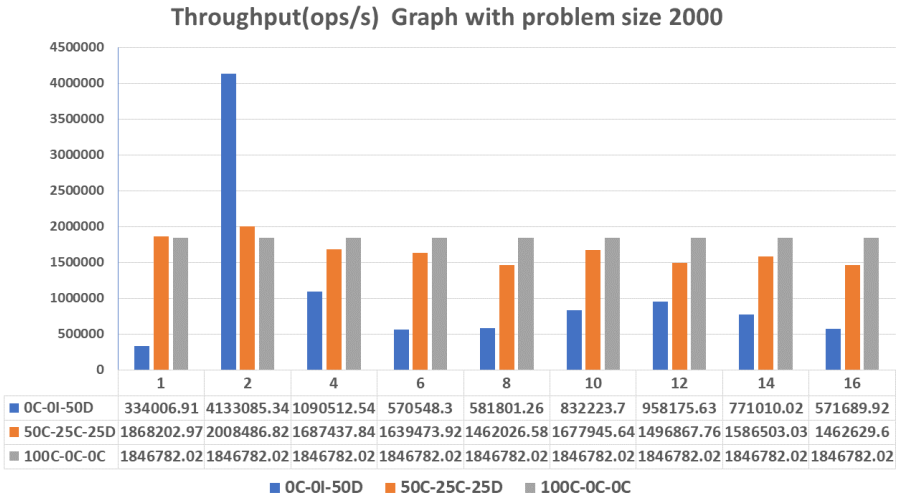


Figure 4: Fine-Grain Throughput of 2000 problem size on different threads and workload

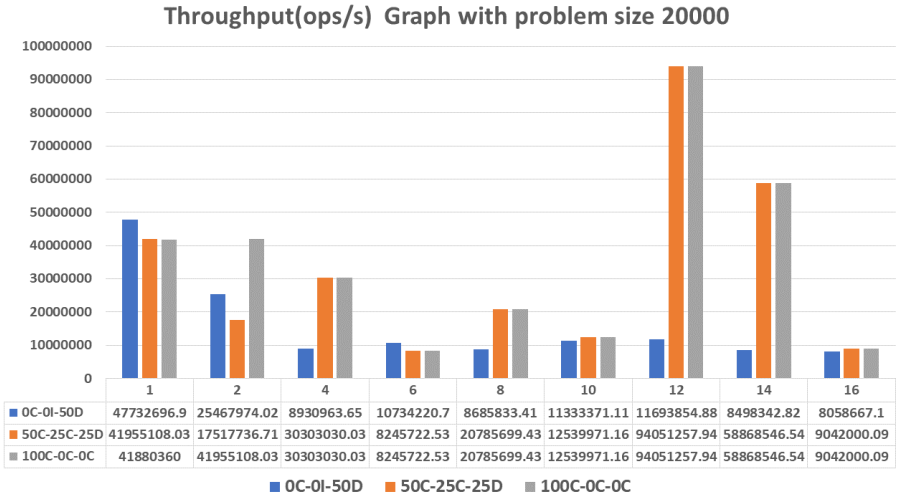


Figure 5: Fine-Grain Throughput of 20000 problem size on different threads and workload

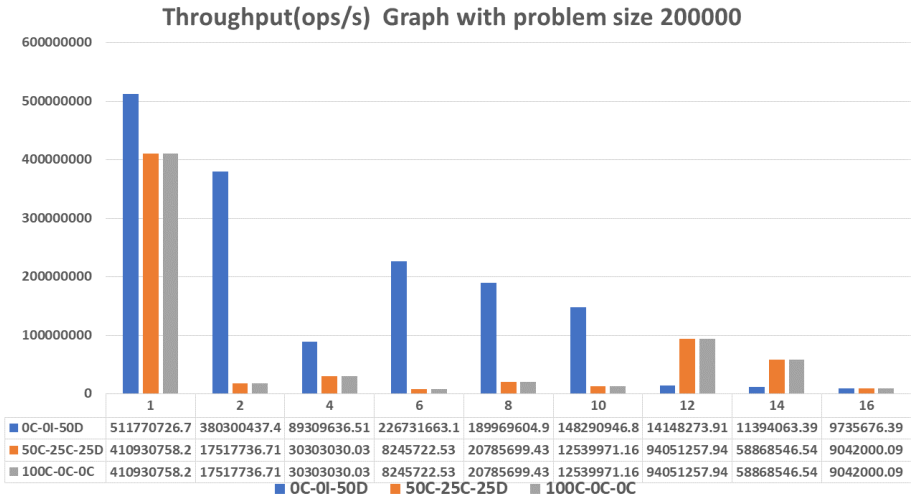


Figure 6: Fine-Grain Throughput of 200000 problem size on different threads and workload

3 Optimistic Synchronization

Optimistic synchronization applied to a doubly linked list is a concurrency control technique that, similar to fine-grain synchronization, aims to enable concurrent operations while minimizing the use of locks. It operates on the assumption that conflicts between threads are rare and validates operations' correctness only if necessary. Let's delve deeper into optimistic synchronization with respect to a doubly linked list:

3.1 Optimistic Synchronization for a Doubly Linked List

- **Rare Conflict Assumption**: Optimistic synchronization starts with the assumption that conflicts between threads, such as simultaneous attempts to modify the same part of a doubly linked list, are infrequent. This assumption allows threads to perform operations concurrently without immediate synchronization.
- **Lock-Free Operations**: Threads are permitted to execute operations on the doubly linked list without acquiring locks. For instance, multiple threads can concurrently attempt insertions, deletions, or traversals on the list without blocking each other.
- **Local Recording**: Instead of immediately updating the list, each thread records its intended changes locally. These changes may involve marking nodes for deletion, inserting new nodes, or updating references.

- **Validation Phase**: After performing their respective operations, threads enter a validation phase. During this phase, they examine and verify the correctness of their recorded changes. This typically involves traversing the list to ensure that the recorded modifications are still valid.
- **Commit or Rollback**: If no conflicts or violations are detected during the validation phase, the recorded changes are considered valid and can be committed to the list. If conflicts or inconsistencies are detected, threads may need to roll back their changes and retry the operation with an updated plan.

3.2 Benefits

- **Reduced Lock Contention**: Optimistic synchronization significantly reduces contention for locks because threads are not blocked during their initial operations. This can lead to enhanced concurrency and system throughput.
- **Improved Responsiveness**: Threads can execute operations without waiting for locks, potentially enhancing response times in applications where responsiveness is critical.

Optimistic synchronization applied to a doubly linked list within the context of different problem sizes, thread counts, and workloads can have varying implications on performance and scalability. Let's explore how optimistic synchronization behaves under these conditions:

3.3 Problem Sizes (2×10^3, 2×10^4, and 10^5)

- **Smaller Problem Sizes**: In cases with smaller problem sizes, optimistic synchronization may perform well as conflicts between threads are less likely. With fewer nodes, the likelihood of concurrent access to the same nodes is reduced.
- **Larger Problem Sizes**: As the problem size increases, the potential for conflicts also increases. Optimistic synchronization may still perform adequately as long as the conflicts remain rare. The key advantage is that it allows for greater concurrency in scenarios where conflicts are infrequent.

3.4 Thread Counts (1, 2, 4, 6, 8, 10, 12, 14, and 16)

- **Low Thread Counts**: With a small number of threads, the likelihood of conflicts remains low, and optimistic synchronization can allow for good concurrency. Threads can execute operations concurrently without significant contention.
- **High Thread Counts**: Optimistic synchronization shines in situations with high thread counts. As more threads contend for access to the doubly

linked list, optimistic synchronization enables them to perform operations concurrently, reducing lock contention and enabling better parallelism.

3.5 Workloads (0C-0I-50D, 50C-25I-25D, and 100C-0I-0D)

- **Read-Heavy Workloads (0C-0I-50D)**: Optimistic synchronization may perform well in this scenario as read operations can proceed concurrently without locking. The absence of inserts and updates makes conflicts less likely.
- **Mixed Workloads (50C-25I-25D)**: In workloads with mixed operations, optimistic synchronization can be beneficial. Threads performing insertions and deletions can operate concurrently, and read operations are not blocked.
- **Write-Heavy Workloads (100C-0I-0D)**: Optimistic synchronization can be effective even in write-heavy scenarios if conflicts are rare. However, if conflicts become frequent due to intense write operations, it may lead to more rollbacks and retries, potentially impacting performance.

3.6 Considerations

- **Conflict Detection**: Detecting conflicts during the validation phase is crucial. The effectiveness of optimistic synchronization depends on the accuracy of this conflict detection.
- **Retry Mechanisms**: Proper mechanisms for retrying operations after conflicts are essential to ensure correctness and maintain system responsiveness.
- **Performance Trade-offs**: While optimistic synchronization can provide better concurrency and responsiveness in low-contention scenarios, it may introduce additional complexity and overhead in terms of conflict detection and resolution.

In summary, optimistic synchronization for a doubly linked list allows multiple threads to perform operations concurrently with the assumption that conflicts are rare. It can be effective in scenarios with various problem sizes, thread counts, and workloads, but the performance and effectiveness depend on the specific characteristics of the application and the accuracy of conflict detection. Careful design and tuning are necessary to leverage the benefits of optimistic synchronization effectively.

3.7 Result

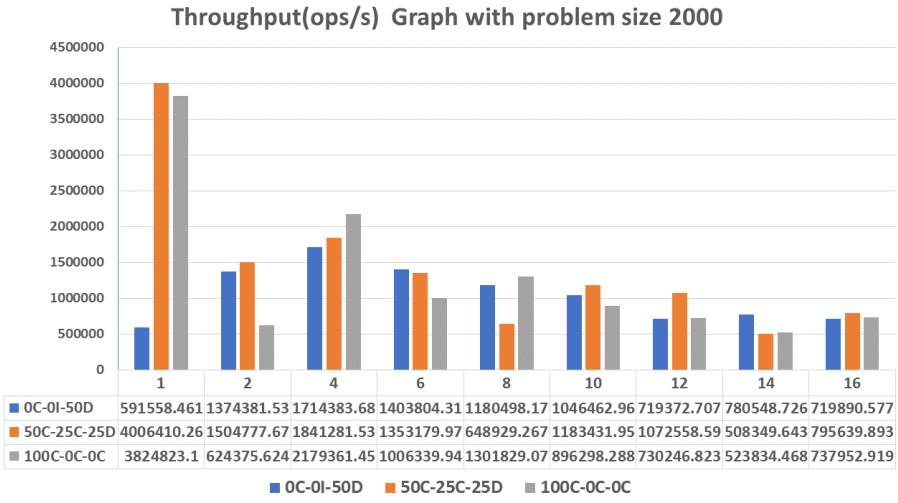


Figure 7: Optimistic Throughput of 2000 problem size on different threads and workload

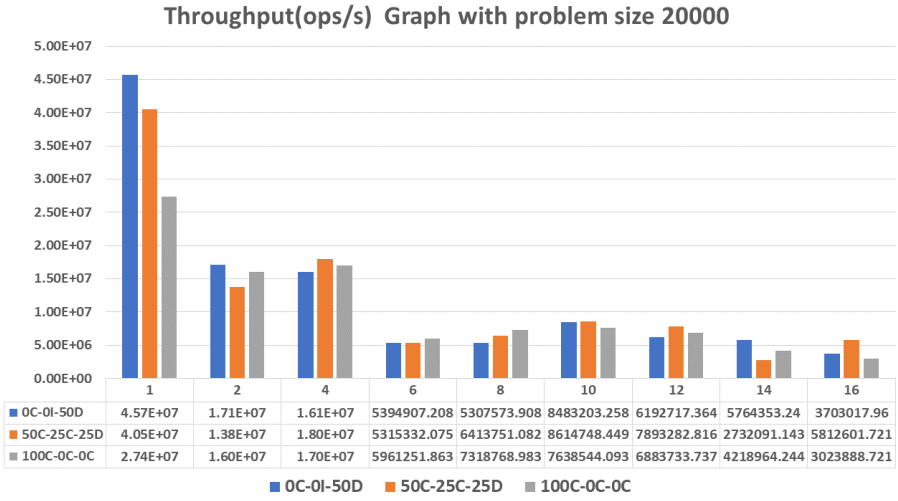


Figure 8: Optimistic Throughput of 20000 problem size on different threads and workload

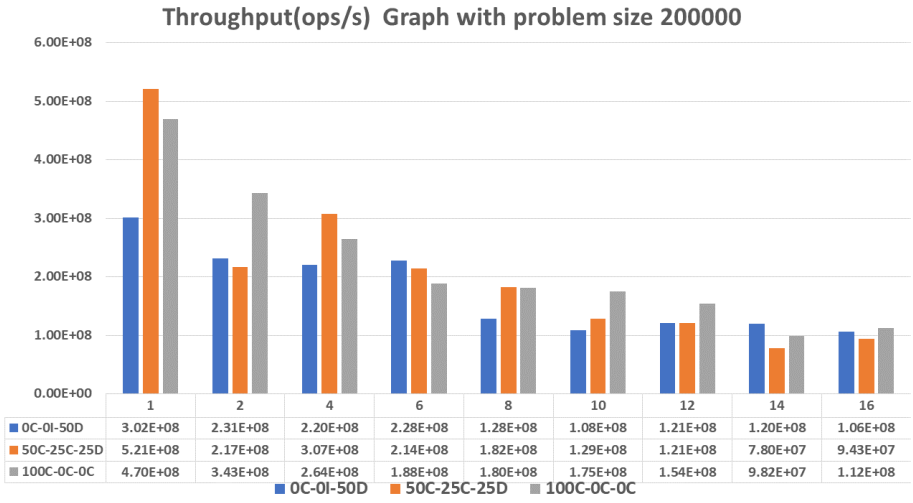


Figure 9: Optimistic of 200000 problem size on different threads and workload

4 Lazy Synchronization

Lazy synchronization, when applied to a doubly linked list, is a concurrency control technique that aims to minimize the use of locks or synchronization mechanisms while allowing multiple threads to operate concurrently. It defers synchronization until the last possible moment, typically during read and write operations. Here's an in-depth explanation of lazy synchronization in the context of a doubly linked list, following a similar pattern as the explanation of fine-grain synchronization:

4.1 Lazy Synchronization for a Doubly Linked List

- **Minimal Lock Usage**: Lazy synchronization avoids the upfront acquisition of locks when reading or writing to a doubly linked list. Instead, it allows multiple threads to perform these operations concurrently without immediate synchronization.
- **Recording Changes**: Threads record their intended changes locally. For instance, when inserting or deleting nodes, a thread may mark the nodes for later modification.
- **Synchronization During Access**: Lazy synchronization introduces synchronization only when necessary. Threads synchronize access to specific nodes or portions of the list just before performing the actual read or write operation.
- **Read Operations**: Threads can read nodes without acquiring locks. They access the nodes directly but may synchronize briefly if another

thread is in the process of updating or deleting the node to ensure data consistency.

- **Write Operations**: When a thread intends to modify a node (e.g., insert, delete, or update), it synchronizes on that node only. This allows the thread to make its modifications safely, ensuring that other threads do not interfere with the update.

4.2 Benefits

- **Reduced Lock Contention**: Lazy synchronization can significantly reduce contention for locks because threads are not blocked during their initial operations. Locks are acquired only when necessary, minimizing the time spent waiting for locks.
- **Improved Concurrency**: Threads can operate concurrently, leading to better overall system performance and responsiveness.

4.3 Challenges and Considerations

- **Data Consistency**: Ensuring data consistency is crucial in lazy synchronization. Developers must carefully manage synchronization points to avoid data corruption or inconsistencies.
- **Correctness**: Proper conflict resolution mechanisms must be in place to handle scenarios where multiple threads attempt to modify the same node simultaneously.
- **Complexity**: Implementing and maintaining lazy synchronization can be more complex compared to fine-grain synchronization because it requires careful management of synchronization points.

Lazy synchronization applied to a doubly linked list within the context of different problem sizes, thread counts, and workloads aims to optimize concurrency while minimizing lock contention. Let's explore how lazy synchronization behaves under these conditions:

4.4 Problem Sizes ($\mathbf{2 \times 10^3}$, $\mathbf{2 \times 10^4}$, and 10^5)

- **Smaller Problem Sizes**: In cases with smaller problem sizes, lazy synchronization may perform well, as conflicts between threads are less likely due to the limited number of nodes. The technique allows multiple threads to operate concurrently with minimal synchronization overhead.
- **Larger Problem Sizes**: As the problem size increases, the potential for conflicts and concurrent access to nodes also increases. Lazy synchronization continues to be effective if the contention for specific nodes remains low.

4.5 Thread Counts (1, 2, 4, 6, 8, 10, 12, 14, and 16)

- **Low Thread Counts**: With a small number of threads, the likelihood of conflicts and contention is generally low. Lazy synchronization allows threads to perform operations concurrently without significant synchronization overhead.
- **High Thread Counts**: Lazy synchronization shines when there are many threads contending for access to the doubly linked list. Threads can read and write concurrently with minimal lock contention, leading to improved concurrency and responsiveness.

4.6 Workloads (0C-0I-50D, 50C-25I-25D, and 100C-0I-0D)

- **Read-Heavy Workloads (0C-0I-50D)**: Lazy synchronization can be effective in read-heavy scenarios because it minimizes the need for locks during reads. Threads can read nodes concurrently without blocking, leading to efficient read operations.
- **Mixed Workloads (50C-25I-25D)**: In workloads with mixed operations, lazy synchronization allows concurrent reads and minimizes locking overhead during insertions and deletions. This approach can maintain good performance and responsiveness.
- **Write-Heavy Workloads (100C-0I-0D)**: Lazy synchronization can still be advantageous in write-heavy scenarios if conflicts are infrequent. By minimizing lock usage during reads and only synchronizing when necessary during writes, it can reduce lock contention and support efficient write operations.

4.7 Considerations

- **Conflict Resolution**: Proper conflict resolution mechanisms must be in place to handle situations where multiple threads attempt to modify the same node simultaneously.
- **Data Consistency**: Ensuring data consistency is crucial. Developers must manage synchronization points effectively to prevent data corruption or inconsistencies.
- **Complexity**: Implementing and maintaining lazy synchronization can be more complex compared to other synchronization methods because it requires careful management of synchronization points.

In summary, lazy synchronization for a doubly linked list allows multiple threads to perform read and write operations concurrently with minimal locking overhead. Its effectiveness depends on the specific application, problem size, thread count, and workload characteristics. It is particularly useful when contention

for specific nodes is low, making it a suitable choice for optimizing concurrency and responsiveness in certain scenarios.

4.8 Result

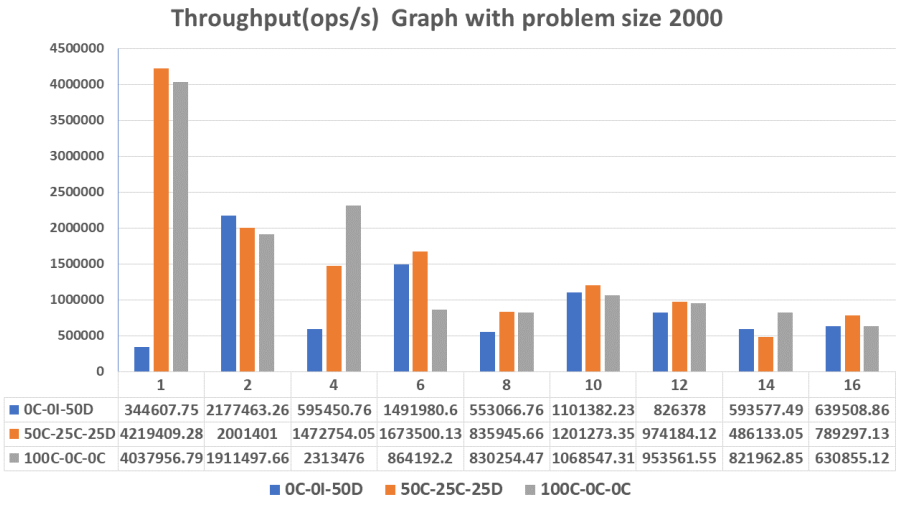


Figure 10: Optimistic Throughput of 2000 problem size on different threads and workload

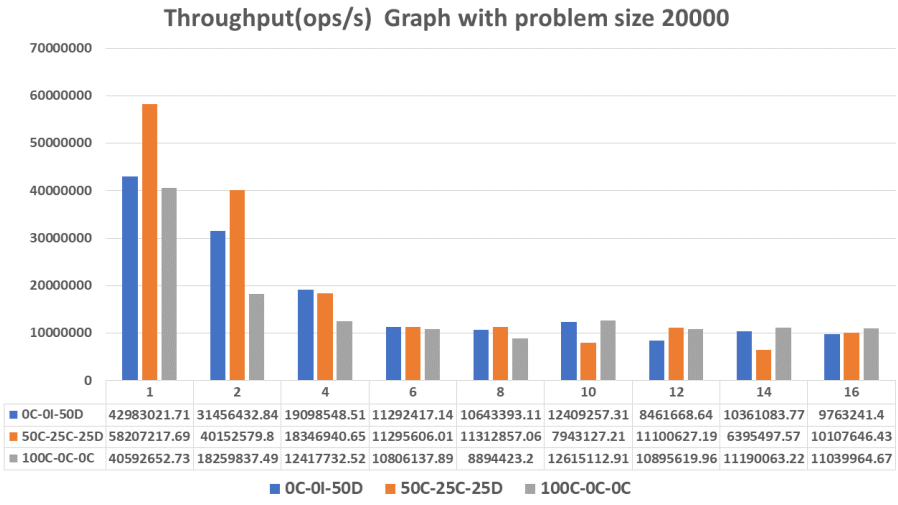


Figure 11: Optimistic Throughput of 20000 problem size on different threads and workload

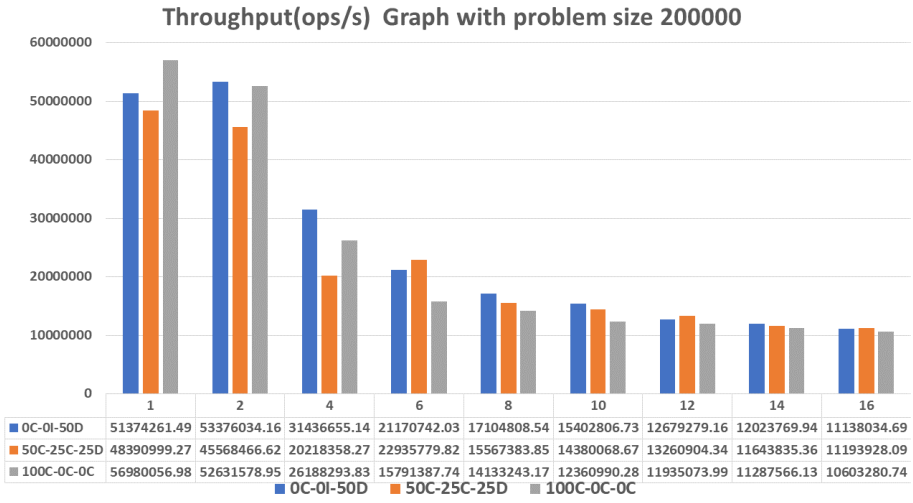


Figure 12: Optimistic Throughput of 200000 problem size on different threads and workload

5 Non-blocking Synchronization

Non-blocking synchronization, when applied to a doubly linked list, is a concurrency control technique that enables multiple threads to operate concurrently without using locks. It guarantees that at least one thread will make progress even in the presence of contention or thread failures. It relies on atomic operations and is designed to avoid the problems of contention and blocking that can occur with traditional lock-based synchronization. Here's an in-depth explanation of non-blocking synchronization in the context of a doubly linked list, following a similar pattern as the explanation of fine-grain synchronization:

5.1 Non-blocking Synchronization for a Doubly Linked List

- **Lock-Free and Wait-Free**: Non-blocking synchronization aims to be both lock-free and wait-free. Lock-free means that even in the presence of contention, some thread will always make progress. Wait-free means that each thread is guaranteed to complete its operation in a bounded number of steps.
- **Atomic Operations**: Non-blocking synchronization relies on atomic operations provided by the hardware or programming language to ensure that shared data structures like the doubly linked list are accessed safely. These atomic operations are indivisible and cannot be interrupted by other threads.
- **No Locks**: Unlike traditional locks, non-blocking synchronization avoids blocking threads. Threads can operate concurrently without waiting for a lock to be released. If a thread encounters contention, it uses atomic operations to resolve conflicts without blocking.
- **Retry Mechanisms**: In non-blocking algorithms, threads perform operations with the expectation that they may fail due to contention. If an operation fails (e.g., another thread modified the same node), the failing thread retries its operation until it succeeds. This retry mechanism ensures that progress is always made.
- **Shared Data Structures**: Non-blocking synchronization is particularly useful for shared data structures like doubly linked lists. Threads can insert, delete, and traverse nodes in the list concurrently without traditional locking, as long as atomic operations are used to maintain data consistency.

5.2 Benefits

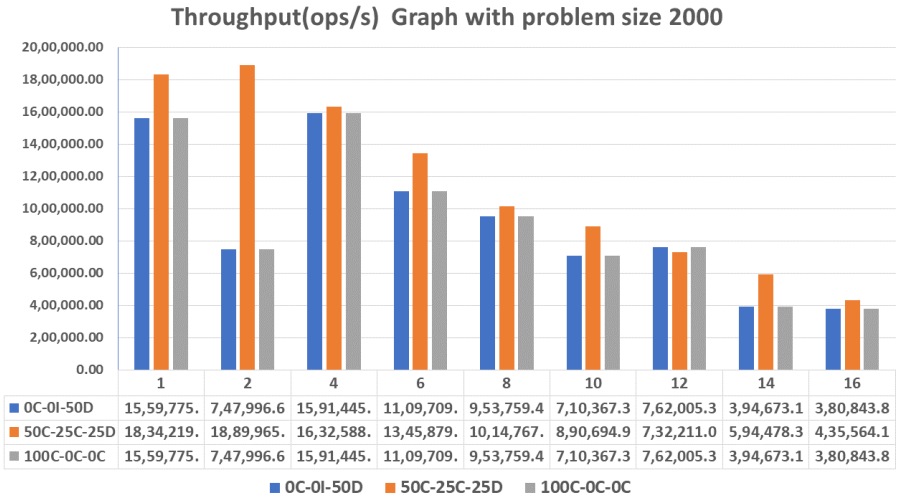
- **High Concurrency**: Non-blocking synchronization allows for high levels of concurrency, making it suitable for multi-core processors and systems with many threads.

- **No Deadlocks**: Since threads do not block while waiting for locks, the possibility of deadlocks is eliminated.
- **Progress Guarantees**: Non-blocking algorithms provide progress guarantees, ensuring that threads continue to make progress even in the presence of contention.

5.3 Challenges and Considerations

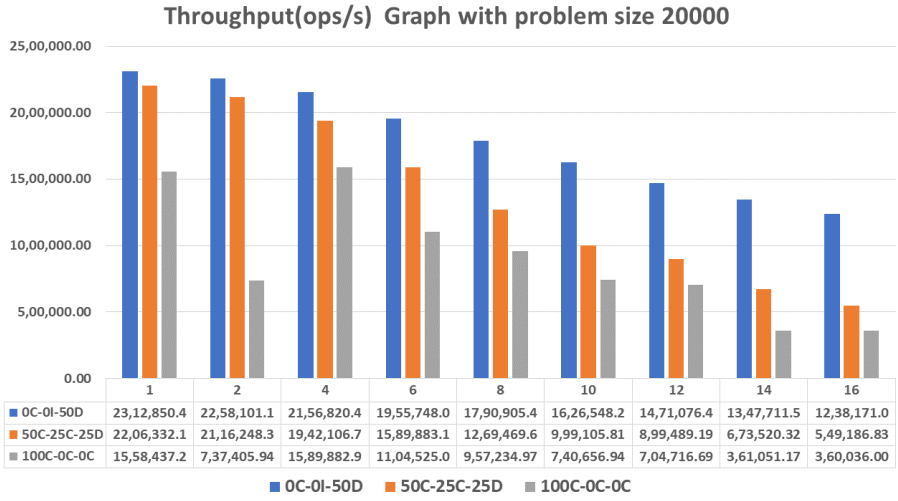
- **Complexity**: Non-blocking algorithms can be complex to design and implement due to the need for atomic operations and retry mechanisms.
- **Performance Trade-offs**: While non-blocking synchronization can provide high concurrency, it may come with performance overhead due to retries and atomic operations.
- **Correctness**: Ensuring the correctness of non-blocking algorithms can be challenging, and thorough testing is required to verify their behavior under various conditions.

5.4 Results



	1	2	4	6	8	10	12	14	16
0C-0I-50D	15,59,775.	7,47,996.6	15,91,445.	11,09,709.	9,53,759.4	7,10,367.3	7,62,005.3	3,94,673.1	3,80,843.8
50C-25C-25D	18,34,219.	18,89,965.	16,32,588.	13,45,879.	10,14,767.	8,90,694.9	7,32,211.0	5,94,478.3	4,35,564.1
100C-0C-0C	15,59,775.	7,47,996.6	15,91,445.	11,09,709.	9,53,759.4	7,10,367.3	7,62,005.3	3,94,673.1	3,80,843.8

Figure 13: Non-blocking Throughput of 2000 problem size on different threads and workload



	1	2	4	6	8	10	12	14	16
0C-0I-50D	23,12,850.4	22,58,101.1	21,56,820.4	19,55,748.0	17,90,905.4	16,26,548.2	14,71,076.4	13,47,711.5	12,38,171.0
50C-25C-25D	22,06,332.1	21,16,248.3	19,42,106.7	15,89,883.1	12,69,469.6	9,99,105.81	8,99,489.19	6,73,520.32	5,49,186.83
100C-0C-0C	15,58,437.2	7,37,405.94	15,89,882.9	11,04,525.0	9,57,234.97	7,40,656.94	7,04,716.69	3,61,051.17	3,60,036.00

Figure 14: Non-blocking Throughput of 20000 problem size on different threads and workload

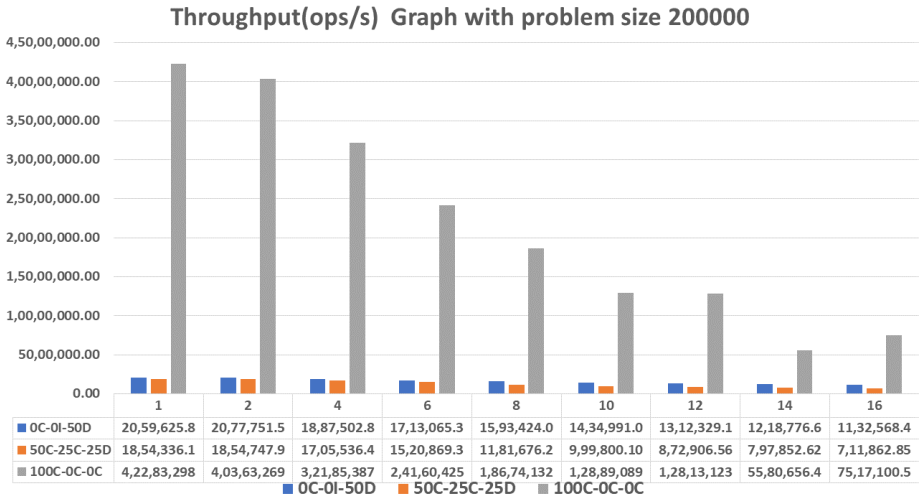


Figure 15: Non-blocking Throughput of 200000 problem size on different threads and workload

6 Source Code Execution

The steps to execute the source code for this assignments is mentioned below:

- **IDE** : Netbeans
- **Synchronization Technique**
  - Coarse-grain Synchronization
  - Fine-grain Synchronization
  - Optimistic Synchronization
  - Lazy Synchronization
- **Uncomment Line** : In Main Function uncomment the line in as per the Synchronization Technique which you want to run**Line Number:** as well the object**Line Number:**
- **Input Argument** : Input the thread count, Time, Range, No Contains Operations, No Insert Operation and No Delete Operation
- **Compilation**: Compile the code
- **Execute**: Execute the buid/compiled code
- **Synchronization Technique**
- **IDE** : Netbeans

  - Non-blocking Synchronization

- **Compilation**: Compile the code
- **Execute**: Execute the buid/compiled code

References

- The Art of Multiprocessor Programming by Maurice Herlihy and Nir Shavit.
- Concurrency in Practice" by Brian Goetz.