Global Journal of Computer Science and Technology: C
Software & Data Engineering
Volume 14 Issue 3 Version 1.0 Year 2014
Type: Double Blind Peer Reviewed International Research Journal
Publisher: Global Journals Inc. (USA)
Online ISSN: 0975-4172 & Print ISSN: 0975-4350

Concurrent Access Algorithms for Different Data Structures: A Research Review

By Ms. Ranjeet Kaur & Dr. Pushpa Rani Suri

Kurukshetra University, India

Abstract - Algorithms for concurrent data structure have gained attention in recent years as multi-core processors have become ubiquitous. Several features of shared-memory multiprocessors make concurrent data structures significantly more difficult to design and to verify as correct than their sequential counterparts. The primary source of this additional difficulty is concurrency. This paper provides an overview of the some concurrent access algorithms for different data structures.

Keywords: concurrency, lock-free, non-blocking, mem-ory management, compares and swap, elimination.

GJCST-C Classification : E.1

Concurrent Access Algorithms for Different Data Structures: A Research Review

Ms. Ranjeet Kaur [α] & Dr. Pushpa Rani Suri [σ]

Abstract- Algorithms for concurrent data structure have gained attention in recent years as multi-core processors have become ubiquitous. Several features of shared-memory multiprocessors make concurrent data structures significantly more difficult to design and to verify as correct than their sequential counterparts. The primary source of this additional difficulty is concurrency. This paper provides an overview of the some concurrent access algorithms for different data structures.

Keywords: concurrency, lock-free, non-blocking, memory management, compares and swap, elimination.

I. Introduction

A concurrent data structure is a particular way of storing and organizing data for access by multiple computing threads (or processes) on a computer. The proliferation of commercial shared-memory multiprocessor machines has brought about significant changes in the art of concurrent programming. Given current trends towards low cost chip multithreading (CMT), such machines are bound to become ever more widespread. Shared-memory multiprocessors are systems that concurrently execute multiple threads of computation which communicate and synchronize through data structures in shared memory. Designing concurrent data structures and ensuring their correctness is a difficult task, significantly more challenging than doing so for their sequential counterparts. The difficult of concurrency is aggravated by the fact that threads are asynchronous since they are subject to page faults, interrupts, and so on. To manage the difficulty of concurrent programming, multithreaded applications need synchronization to ensure thread-safety by coordinating the concurrent accesses of the threads. At the same time, it is crucial to allow many operations to make progress concurrently and complete without interference in order to utilize the parallel processing capabilities of contemporary architectures. The traditional way to implement shared data structures is to use mutual exclusion (locks) to ensure that concurrent operations do not interfere with one another. Locking has a number of disadvantages with respect to software engineering, fault-tolerance, and scalability. In response, researchers have investigated a variety of alternative synchronization techniques that do not employ mutual exclusion. A synchronization technique is wait-free if it ensures that every thread will continue to make progress in the face of arbitrary delay (or even failure) of other threads. It is lock-free if it ensures only that some thread always makes progress. While wait-free synchronization is the ideal behavior (thread starvation is unacceptable), lock-free synchronization is often good enough for practical purposes (as long as starvation, while possible in principle, never happens in practice).The synchronization primitives provided by most modern architectures, such as compare-and-swap (CAS) or load-locked/store-conditional (LL/SC) are powerful enough to achieve wait-free (or lock-free) implementations of any linearizable data object [23]. The remaining paper will discussed about the different data structures, concurrency control methods and various techniques given for the concurrent access to these data structures.

II. Data Structures

Data can be organized in many ways and a data structure is one of these ways. It is used to represent data in the memory of the computer so that the processing of data can be done in easier way. In other words, data structures are the logical and mathematical model of a particular organization of data. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers. A data structure can be broadly classified into (i) Primitive data structure (ii) Non-primitive data structure.

Primitive Data Structure: The data structures, typically those data structure that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float.

Non-Primitive Data Structure: The data structures, which are not primitive, are called non-primitive data structures. There are two types of-primitive data structures.

a) *Linear Data Structures*

A list, which shows the relationship of adjacency between elements, is said to be linear data structure. The most, simplest linear data structure is a 1-D array, but because of its deficiency, list is frequently used for different kinds of data.

Author α σ: Kurukshetra University, Kurukshetra.
e-mails: kaurranjeet 2203@gmail.com, pushpa.suri@yahoo.com

A [0]	A[1]	A[2]	A[3]	A[4]	A[5]

Figure 1 : A 1-D Array of 6 Elements.

b) Non-Linear Data Structure

A list, which doesn't show the relationship of adjacency between elements, is said to be non-linear data structure.

i. *Linear Data Structure*

A list is an ordered list, which consists of different data items connected by means of a link or pointer. This type of list is also called a linked list. A linked list may be a single list or double linked list.

- *Single linked list:* A single linked list is used to traverse among the nodes in one direction.

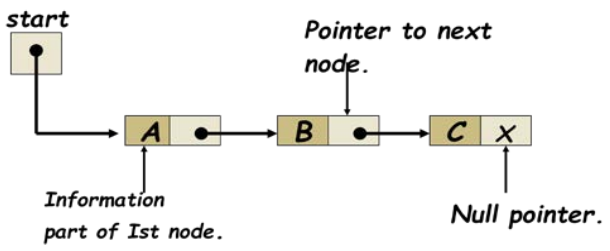

Figure 2 : A single three Nodes linked list.

Double linked list: A double linked list is used to traverse among the nodes in both the directions.

A list has two subsets. They are: -

Stack: It is also called as last-in-first-out (LIFO) system. It is a linear list in which insertion and deletion take place only at one end. It is used to evaluate different expressions.

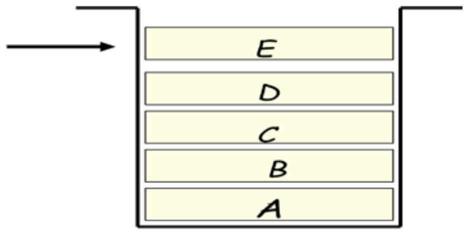

Figure 3 : A Stack with Elements.

- *Queue:* It is also called as first-in-first-out (FIFO) system. It is a linear list in which insertion takes place at once end and deletion takes place at other end. It is generally used to schedule a job in operating systems and networks.

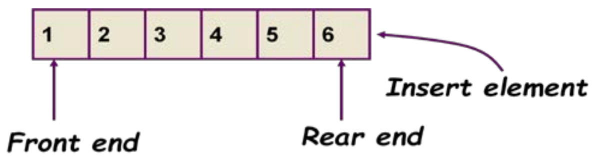

Figure 4 : A Queue with 6 Elements.

ii. *Non-Linear Data Structure*

The frequently used non-linear data structures are

- *Trees:* It maintains hierarchical relationship between various elements

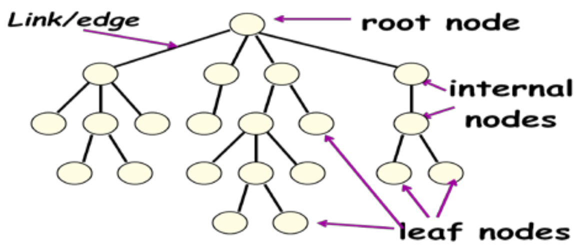

Figure 5 : A Binary Tree.

- *Graphs:* It maintains random relationship or point-to-point relationship between various elements.

III. Concurrency Control

Simultaneous execution of multiple threads/process over a shared data structure access can create several data integrity and consistency problems:

- Lost Updates.
- Uncommitted Data.
- Inconsistent retrievals

All above are the reasons for introducing the concurrency control over the concurrent access of shared data structure. Concurrent access to data structure shared among several processes must be synchronized in order to avoid conflicting updates. Synchronization is referred to the idea that multiple processes are to join up or handshake at a certain points, in order to reach agreement or commit to a certain sequence of actions. The thread synchronization or serialization strictly defined is the application of particular mechanisms to ensure that two concurrently executing threads or processes do not execute specific portions of a program at the same time. If one thread has begun to execute a serialized portion of the program, any other thread trying to execute this portion must wait until the first thread finishes.

Concurrency control techniques can be divided into two categories.

- Blocking
- Non-blocking

Both of these are discussed in below sub-sections.

a) Blocking

Blocking algorithms allow a slow or delayed process to prevent faster processes from completing operations on the shared data structure indefinitely. On asynchronous (especially multiprogrammed) multiprocessor systems, blocking algorithms suffer significant performance degradation when a process is halted or delayed at an inopportune moment. Many of the existing

Global Journal of Computer Science and Technology (C) Volume XIV Issue III Version I Year 2014

concurrent data structure algorithms that have been developed use mutual exclusion i.e. some form of locking.

Mutual exclusion degrades the system's overall performance as it causes blocking, due to that other concurrent operations cannot make any progress while the access to the shared resource is blocked by the lock. The limitation of blocking approach are given below

- *Priority Inversion:* occurs when a high-priority process requires a lock holded by a lower-priority process.
- *Convoying:* occurs when a process holding a lock is rescheduled by exhausting its quantum, by a page fault or by some other kind of interrupt. In this case, running processes requiring the lock are unable to progress.
- *Deadlock:* can occur if different processes attempt to lock the same set of objects in different orders.
- Locking techniques are not suitable in a real-time context and more generally, they suffer significant performance degradation on multiprocessors systems.

b) Non-Blocking

Non-blocking algorithm Guarantees that the data structure is always accessible to all processes and an inactive process cannot render the data structure inaccessible. Such an algorithm ensures that some active process will be able to complete an operation in a finite number of steps making the algorithm robust with respect to process failure [22]. In the following sections we discuss various non-blocking properties with different strength.

- *Wait-Freedom:* A method is wait-free if every call is guaranteed to finish in a finite number of steps. If a method is bounded wait-free then the number of steps has a finite upper bound, from this definition it follows that wait-free methods are never blocking, therefore deadlock cannot happen. Additionally, as each participant can progress after a finite number of steps (when the call finishes), wait-free methods are free of starvation.
- *Lock-Freedom:* Lock-freedom is a weaker property than wait-freedom. In the case of lock-free calls, infinitely often some method finishes in a finite number of steps. This definition implies that no deadlock is possible for lock-free calls. On the other hand, the guarantee that some call finishes in a finite number of steps is not enough to guarantee that all of them eventually finish. In other words, lock-freedom is not enough to guarantee the lack of starvation.
- *Obstruction-Freedom:* Obstruction-freedom is the weakest non-blocking guarantee discussed here. A method is called obstruction-free if there is a point in time after which it executes in isolation (other threads make no steps, e.g.: become suspended), it finishes in a bounded number of steps. All lock-free objects are obstruction-free, but the opposite is generally not true. Optimistic concurrency control (OCC) methods are usually obstruction-free. The OCC approach is that every participant tries to execute its operation on the shared object, but if a participant detects conflicts from others, it rolls back the modifications, and tries again according to some schedule. If there is a point in time, where one of the participants is the only one trying, the operation will succeed.

In the sequential setting, data structures are crucially important for the performance of the respective computation. In the parallel programming setting, their importance becomes more crucial because of the increased use of data and resource sharing for utilizing parallelism. In parallel programming, computations are split into subtasks in order to introduce parallelization at the control/computation level. To utilize this opportunity of concurrency, subtasks share data and various resources (dictionaries, buffers, and so forth). This makes it possible for logically independent programs to share various resources and data structures.

Concurrent data structure designers are striving to maintain consistency of data structures while keeping the use of mutual exclusion and expensive synchronization to a minimum, in order to prevent the data structure from becoming a sequential bottleneck. Maintaining consistency in the presence of many simultaneous updates is a complex task. Standard implementations of data structures are based on locks in order to avoid inconsistency of the shared data due to concurrent modifications. In simple terms, a single lock around the whole data structure may create a bottleneck in the program where all of the tasks serialize, resulting in a loss of parallelism because too few data locations are concurrently in use. Deadlocks, priority inversion, and convoying are also side-effects of locking. The risk for deadlocks makes it hard to compose different blocking data structures since it is not always possible to know how closed source libraries do their locking. Lock-free implementations of data structures support concurrent access. They do not involve mutual exclusion and make sure that all steps of the supported operations can be executed concurrently. Lock-free implementations employ an optimistic conflict control approach, allowing several processes to access the shared data object at the same time. They suffer delays only when there is an actual conflict between operations that causes some operations to retry. This feature allows lock-free algorithms to scale much better when the number of processes increases. An implementation of a data structure is called lock-free if it allows multiple processes/threads to access the data structure concurrently and also guarantees that at least one operation among those finishes in a finite number of its own steps regardless of the state of the other operations. A consistency (safety) requirement for lock-free data structures is linearizability [24], which ensures that each operation on the data appears to take effect instantaneously during its actual duration and the effect of all operations are consistent with the object's sequential specification. Lock-free data structures offer several advantages over their blocking counterparts, such as being immune to deadlocks, priority inversion, and convoying, and have been shown to work well in practice in many different settings [26, 25].

The remaining paper will explore the access of different data structures like stack, queue, trees, priority queue, and linked list in concurrent environment. How the sequence of data structure operations changes during concurrent access. These techniques will be based on blocking and non-blocking.

IV. Literature Review

a) Stack Data Structure

Stack is the simplest sequential data structures. Numerous issues arise in designing concurrent versions of these data structures, clearly illustrating the challenges involved in designing data structures for shared-memory multiprocessors. A concurrent stack is a data structure linearizable to a sequential stack that provides push and pop operations with the usual LIFO semantics. Various alternatives exist for the behavior of these data structures in full or empty states, including returning a special value indicating the condition, raising an exception, or blocking.

There are several lock-based concurrent stack implementations in the literature. Typically, lock-based stack algorithms are expected to offer limited robustness.

The first non-blocking implementation of concurrent link based stack was first proposed by Trieber et al [1]. It represented the stack as a singly linked list with a top pointer. It uses compare-and-swap to modify the value of Top atomically. However, this stack was very simple and can be expected to be quite efficient, but no performance results were reported for nonblocking stacks. When Michael et. al [2] compare the performance of Treiber's stack to an optimized nonblocking algorithm based on Herlihy's methodology [28], and several lock-based stacks such as an MCS lock in low load situations[29]. They concluded that Treiber's algorithm yields the best overall performance, but this performance gap increases as the degree of multiprogramming grows. All this happen due to contention and an inherent sequential bottleneck.

Hendler et al. [3] observe that any stack implementation can be made more scalable using the elimination technique [23]. Elimination allows pairs of operations with reverse semantics like pushes and pops on a stack-to complete without any central coordination, and therefore substantially aids scalability. The idea is that if a pop operation can find a concurrent push operation to "partner" with, then the pop operation can take the push operation's value, and both operations can return immediately.

b) Queue Data Structure

A concurrent queue is a data structure that provides enqueue and dequeue operations with the usual FIFO semantics. Valois et.al [4] presented a list-based nonblocking queue. The represented algorithm allows more concurrency by keeping a dummy node at the head (dequeue end) of a singly linked list, thus simplifying the special cases associated with empty and single-item. Unfortunately, the algorithm allows the tail pointer to lag behind the head pointer, thus preventing dequeuing processes from safely freeing or reusing dequeued nodes. If the tail pointer lags behind and a process frees a dequeued node, the linked list can be broken, so that subsequently enqueued items are lost. Since memory is a limited resource, prohibiting memory reuse is not an acceptable option. Valois therefore proposed a special mechanism to free and allocate memory. The mechanism associates a reference counter with each node. Each time a process creates a pointer to a node it increments the node's reference counter atomically. When it does not intend to access a node that it has accessed before, it decrements the associated reference counter atomically. In addition to temporary links from process-local variables, each reference counter reflects the number of links in the data structure that point to the node in question. For a queue, these are the head and tail pointers and linked-list links. A node is freed only when no pointers in the data structure or temporary variables point to it. Drawing ideas from the previous authors, Michel et.al [5] presented a new non-blocking concurrent queue algorithm, which is simple, fast, and practical. The algorithm implements the queue as a singly-linked list with Head and Tail pointers. Head always points to a dummy node, which is the first node in the list. Tail points to either the last or second to last node in the list. The algorithm uses compare and swap, with modification counters to avoid the ABA problem. To allow dequeuing processes to free dequeue nodes, the dequeue operation ensures that Tail does not point to the dequeued node nor to any of its predecessors. This means that dequeued nodes may safely be re-used.

The Mark et al [6] introduced a scaling technique for queue data structure which was earlier applied to LIFO data structures like stack. They transformed existing nonscalable FIFO queue implementations into scalable implementations using the elimination technique, while preserving lock-freedom and linearizability.

In all previously FIFO queue algorithms, concurrent Enqueue and Dequeue operations synchronized on a small number of memory locations, such algorithms can only allow one Enqueue and one Dequeue operation to complete in parallel, and therefore cannot scale to large numbers of concurrent operations. In the LIFO structures elimination works by allowing opposing operations such as pushes and pops to exchange values in a pair wise distributed fashion without synchronizing on a centralized data structure. This technique was straightforward in LIFO ordered structures [23]. However, this approach seemingly contradicts in a queue data structure, a Dequeue operation must take the oldest value currently waiting in the queue. It apparently cannot eliminate with a concurrent Enqueue. For example, if a queue contains a single value 1, then after an Enqueue of 2 and a Dequeue, the queue contains 2, regardless of the order of these operations.

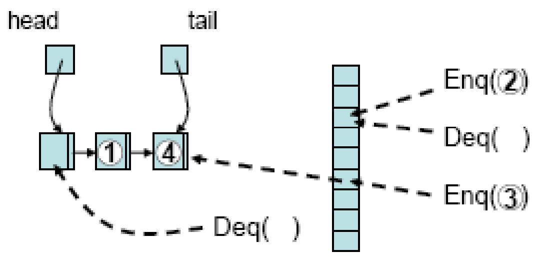

Figure 6 : Shows an Example Execution

Thus, because the queue changes, we cannot simply eliminate the Enqueue and Dequeue. In a empty queue , we could eliminate an Enqueue-Dequeue pair, because in this case the queue is unchanged by an Enqueue immediately followed by a Dequeue. In case when queue is non empty , we must be aware with linearizability correctness condition [24,25], which requires that we can order all operations in such a way that the operations in this order respect the FIFO queue semantics, but also so that no process can detect that the operations did not actually occur in this order. If one operation completes before another begins, then we must order them in this order. Otherwise, if the two are concurrent, we are free to order them however we wish. Key to their approach was the observation that they wanted to use elimination when the load on the queue is high. In such cases, if an Enqueue operation is unsuccessful in an attempt to access the queue, it will generally back off before retrying. If in the meantime all values that were in the queue when the Enqueue began are dequeued, then we can "pretend" that the Enqueue did succeed in adding its value to the tail of the queue earlier, and that it now has reached the head and can be dequeued by an eliminating Dequeue. Thus, they used time spent backing off to "age" the unsuccessful Enqueue operations so that they become "ripe" for elimination. Because this time has passed, we ensure that the Enqueue operation is concurrent with Enqueue operations that succeed on the central queue, and this allows us to order the Enqueue before some of them, even though it never succeeds on the central queue. The key is to ensure that Enqueues are eliminated only after sufficient aging.

c) Linked List Data Structure

Implementing linked lists efficiently is very important, as they act as building blocks for many other data structures. The first implementation designed for lock-free linked lists was presented by Valois et .al [19]. The main idea behind this approach was to maintain auxiliary nodes in between normal nodes of the list in order to resolve the problems that arise because of interference between concurrent operations. Also, each node in his list had a backlink pointer which was set to point to the predecessor when the node was deleted. These backlinks were then used to backtrack through the list when there was interference from a concurrent deletion. Another lock-free implementation of linked lists was given by Harris et. al[20]. His main idea was to mark a node before deleting it in order to prevent concurrent operations from changing its right pointer. The previous approach was simpler than later one. Yet another implementation of a lock-free linked list was proposed by Michael [21]. The represented Technique used [20] design to implement the lock free linked list structure. The represented algorithm was compatible with efficient memory management techniques unlike [20] algorithm.

d) Tree Data Structure

A concurrent implementation of any search tree can be achieved by protecting it using a single exclusive lock. Concurrency can be improved somewhat by using a reader-writer lock to allow all read-only (search) operations to execute concurrently with each other while holding the lock.

Kung and Lehman et al. [7] presented a concurrent binary search tree implementation in which update operations hold only a constant number of node locks at a time, and these locks only exclude other update operations: search operations are never blocked. However, this implementation makes no attempt to keep the search tree balanced.

In the context of B+-trees Lehman et al.[8] has expanded some of the ideas of previous technique. The algorithm has property that any process for manipulating the tree uses only a small number of locks at any time, no search through the tree is ever prevented from reading any node, for that purpose they have considered a variant of B* -Tree called Blink- tree.

The Blink-tree is a B*-tree modified by adding a single "link" pointer field to each node This link field points to the next node at the same level of the tree as the current node, except that the link pointer of the rightmost node on a level is a null pointer. This definition

Global Journal of Computer Science and Technology (C) Volume XIV Issue III Version I Year 2014

for link pointers is consistent, since all leaf nodes lie at the same level of the tree. The Blink-tree has all of the nodes at a particular level chained together into a linked list.

In fact, in [8] algorithm, update operations as well as search operations use the lock coupling technique so that no operation ever holds more than two locks at a time, which significantly improves concurrency. This technique has been further refined, so that operations never hold more than one lock at a time [9]. The presented algorithm not addressed how nodes can be merged, instead allowing delete operations to leave nodes underfull. They argue that in many cases delete operations are rare, and that if space utilization becomes a problem, the tree can occasionally be reorganized in “batch” mode by exclusively locking the entire tree. Lanin et al. [10] incorporate merging into the delete operations, similarly to how insert operations split overflowed nodes in previous implementations. Similar to [8] technique, these implementations use links to allow recovery by operations that have mistakenly reached a node that has been evacuated due to node merging. In all of the algorithms discussed above, the maintenance operations such as node splitting and merging (where applicable) are performed as part of the regular update operations.

e) Priority Queue Data Structure

The Priority Queue abstract data type is a collection of items which can efficiently support finding the item with the highest priority. Basic operations are Insert (add an item), FindMin (finds the item with minimum (or maximum) priority), and DeleteMin (removes the item with minimum (or maximum) priority). DeleteMin returns the item removed.

- *Heap-Based Priority Queues:* Many of the concurrent priority queue constructions in the literature are linearizable versions of the heap structures. Again, the basic idea is to use fine-grained locking of the individual heap nodes to allow threads accessing different parts of the data structure to do so in parallel where possible. A key issue in designing such concurrent heaps is that traditionally insert operations proceed from the bottom up and delete-min operations from the top down, which creates potential for deadlock. Biswas et al. [11] present such a lock-based heap algorithm assuming specialized “cleanup” threads to overcome deadlocks. Rao et al. [12] suggest to overcome the drawbacks of [11] using an algorithm that has both insert and delete-min operations proceed from the top down. Ayani et.al [13] improved on their algorithm by suggesting a way to have consecutive insertions be performed on opposite sides of the heap. Hunt et al. [14] present a heap based algorithm that overcomes many of the limitations of the above schemes, especially the need to acquire multiple locks along the traversal path in the heap. It proceeds by locking for a short duration a variable holding the size of the heap and a lock on either the first or last element of the heap. In order to increase parallelism, insertions traverse the heap bottom-up while deletions proceed top-down, without introducing deadlocks. Insertions also employ a left-right technique as in [13] to allow them to access opposite sides on the heap and thus minimize interference.

Unfortunately, the empirical evidence shows, the performance of [14] does not scale beyond a few tens of concurrent processors. As concurrency increases, the algorithm's locking of a shared counter location, introduces a sequential bottleneck that hurts performance. The root of the tree also becomes a source of contention and a major problem when the number of processors is in the hundreds. In summary, both balanced search trees and heaps suffer from the typical scalability impediments of centralized structures: sequential bottlenecks and increased contention. The solution proposed by lotal et.al [15] is to design concurrent priority queues based on the highly distributed SkipList data structures of Pugh [31, 32].

SkipLists are search structures based on hierarchically ordered linked-lists, with a probabilistic guarantee of being balanced. The basic idea behind SkipLists is to keep elements in an ordered list, but have each record in the list be part of up to a logarithmic number of sub-lists. These sub-lists play the same role as the levels of a binary search structure, having twice the number of items as one goes down from one level to the next. To search a list of N items, O (log N) level lists are traversed, and a constant number of items is traversed per level, making the expected overall complexity of an Insert or Delete operation on a SkipList O(logN). Author introduced the SkipQueue, a highly distributed priority queue based on a simple modification of Pugh's concurrent SkipList algorithm [31]. Inserts in the SkipQueue proceed down the levels as in [31]. For Delete-min, multiple minimal" elements are to be handed out concurrently. This means that one must coordinate the requests, with minimal contention and bottlenecking, even though Delete-mins are interleaved with Insert operations. The solution was as follows, keep a specialized delete pointer which points to the current minimal item in this list. By following the pointer, each Delete-min operation directly traverses the lowest level list, until it finds an unmarked item, which it marks as \deleted." It then proceeds to perform a regular Delete operation by searching the SkipList for the items immediately preceding the item deleted at each level of the list and then redirecting their pointers in order to remove the deleted node.

Sundell et.al [16] given an efficient and practical lock-free implementation of a concurrent priority queue that is suitable for both fully concurrent (large multiprocessor) systems as well as pre-emptive (multi-process) systems. Inspired by [15], the algorithm was based on the randomized Skiplist [28] data structure, but in contrast to [15] it is lock-free. The algorithm was based on the sequential Skiplist data structure invented by Pugh [32]. This structure uses randomization and has a probabilistic time complexity of O(logN) where N is the maximum number of elements in the list. The data structure is basically an ordered list with randomly distributed short-cuts in order to improve search times, In order to make the Skiplist construction concurrent and non-blocking; author used three of the standard atomic synchronization primitives, Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS). To insert or delete a node from the list we have to change the respective set of next pointers. These have to be changed consistently, but not necessary all at once. The solution was to have additional information on each node about its deletion (or insertion) status. This additional information will guide the concurrent processes that might traverse into one partial deleted or inserted node. When we have changed all necessary next pointers, the node is fully deleted or inserted.

- *Tree-Based Priority Pools:* Huang and Weihl et al. [18] and Johnson et al.[17] describe concurrent priority pools: priority queues with relaxed semantics that do not guarantee linearizability of the delete-min operations. Their designs were based on a modified concurrent B+-tree implementation. Johnson introduces a "delete bin" that accumulates values to be deleted and thus reduces the load when performing concurrent delete-min operations.

V. Comparison and Analysis

Data structure	Algorithm	Merits	Demerits
Stack	Systems programming: Coping with parallelism	Simple and can be expected to be quite efficient.	Contention and an inherent sequential bottleneck.
	A scalable lock-free stack algorithm	Due to elimination technique there is high degree of parallelism.	
queue	Implementing Lock-Free queues.	Algorithm no longer needs the snapshot, only intermediate state that the queue can be in is if the tail pointer has not been updated	Required either an unaligned compare & swap or a Motorola like double-compare and swap, both of them are not supported on any architecture.
	Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.	The algorithm was simple, fast and practical .it was the clear algorithm of choice for machine that provides a universal atomic primitive.	Pointers are inserted using costly CAS
	Using elimination to implement scalable and lock-free FIFO queues.	1. Scaling technique allows multiple enqueue and dequeue operations to complete in parallel. 2. The concurrent access to the head and tail of the queue do not interfere with each other as long as the queue is non-empty.	1. The elimination back off queue is practical only for very short queues as in order to keep the correct FIFO queue semantics, the enqueue operation cannot be eliminated unless all previous inserted nodes have been dequeued. 2. scalable in performance as compare to previous one but having high overhead.
Tree	Concurrent manipulation of binary search trees.	Algorithm never blocked the search operations	Search tree is not balanced
	Efficient Locking for Concurrent Operations on B-trees,	Small number of locks used	Expansive locks
	A symmetric concurrent b-tree algorithm	They involved the merging as a part of deletion.	Expansive locking

Priority queue	An efficient algorithm for concurrent priority queue heaps	Allows concurrent insertion and deletion in opposite direction.	The performance does not scale beyond a few tens of concurrent processors.
	Skip list-Based Concurrent Priority Queues	Designed a scalable concurrent priority queue for large scale multi-processor.	Algorithm based on locking approach.
	Fast and Lock-Free Concurrent Priority Queues for Multithread System.	This was a first lock-free approach for concurrent priority queue	
	A highly concurrent priority queue based on the b-link tree	Avoid the serialization bottleneck	Needs node to be locked in order to be rebalance
Linked list	Lock-free linked lists using compare-and-swap	Reduced interference of concurrent operations using backlink nodes	
	A pragmatic implementation of non-blocking linked-lists	For making successful updating of nodes, every node to be deleted was marked	Difficult to implement
	High performance dynamic lock-free hash tables and list-based sets.	Efficient with memory management techniques	Poor in performance.

VI. Conclusion

This paper reviews the different data structures and the concurrency control techniques with respect to different data structures (tree, queue, priority queue). The algorithms are categorized on the concurrency control techniques like blocking and non-blocking. Former based on locks and later one can be lock-free, wait-free or obstruction free. In the last we can see that lock free approach outperforms over locking based approach.

References Références Referencias

1. R. K. Treiber, "Systems programming: Coping with parallelism", "RJ 5118, Almaden Research Center, and ''April 1986.
2. M. Michael and M. Scott. "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared - memory multiprocessors." Journal of Parallel and Distributed Computing, 51(1): 1–26, 1998.
3. D. Hendler, N. Shavit, and L. Yerushalmi. "A scalable lock-free stack algorithm." Technical Report TR-2004-128, Sun Microsystems Laboratories, 2004.
4. J. D. Valois. "Implementing Lock-Free queues." In Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, NV, October 1994.
5. M. M. Michael and M. L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms." 15th ACM Symp. On Principles of Distributed Computing (PODC), May 1996. pp.267 – 275.
6. Mark Moir, Daniel Nussbaum, Ori Shalev, Nir Shavit: "Using elimination to implement scalable and lock-free FIFO queues. " SPAA 2005.
7. H. Kung and P. Lehman. "Concurrent manipulation of binary search trees." ACM Transactions on Programming Languages and Systems, 5:354–382, September 1980.
8. P. Lehman and S. Yao. "Efficient Locking for Concurrent Operations on B-trees", ACM Trans. Database Systems, vol. 6,no. 4, 1981.
9. Y. Sagiv. "Concurrent operations on b-trees with overtaking." Journal of Computer and System Sciences, 33(2):275–296, October 1986.
10. V. Lanin and D. Shasha. "A symmetric concurrent b-tree algorithm." In Proceedings of the Fall Joint Computer Conference 1986, pages 380–389. IEEE Computer Society Press, November 1986.
11. J. Biswas and J. Browne. "Simultaneous update of priority structures." In Proceedings of the 1987 International Conference on Parallel Processing, pages 124–131, August 1987.
12. V. Rao and V. Kumar. "Concurrent access of priority queues." IEEE Transactions on Computers, 37:1657–1665, December 1988.
13. R. Ayani. "LR-algorithm: concurrent operations on priority queues." In Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing, pages 22–25, 1991.
14. G. Hunt, M. Michael, S. Parthasarathy, and M. Scott. "An efficient algorithm for concurrent priority queue heaps." Information Processing Letters, 60(3): 151–157, November 1996.
15. LOTAN, N. SHAVIT. "Skiplist-Based Concurrent Priority Queues", International Parallel and Distributed Processing Symposium, 2000.

16. H.Sundell and P.tsigas. “Fast and Lock-Free Concurrent Priority Queues for Multithread System.”
17. T. Johnson. “A highly concurrent priority queue based on the b-link tree.” Technical Report 91-007, University of Florida, August 1991.
18. Q. Huang and W. Weihl”. An evaluation of concurrent priority queue algorithms. “In IEEE Parallel and Distributed Computing Systems, pages 518–525, 1991.
19. J. D. Valois. “Lock-free linked lists using compare-and-swap.” In Proceedings of the 14th ACM Symposium on Principles of Distributed Computing, pages 214–222, 1995.
20. T. L. Harris. “A pragmatic implementation of non-blocking linked-lists.” In Proceedings of the 15th International Symposium on Distributed Computing, pages 300–314, 2001.
21. M. M. Michael. “High performance dynamic lock-free hash tables and list-based sets.” In Proceedings of the 14th annual ACM Symposium on Parallel Algorithms and Architectures, pages 73–82, 2002.
22. M. Greenwald. “Non-Blocking Synchronization and System Design.” PhD thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, A, 8 1999.
23. N. Shavit and D. Touitou. “Elimination trees and the construction of pools and stacks.” Theory of Computing Systems, 30:645–670, 1997.
24. M. Herlihy. “A methodology for implementing highly concurrent data objects.” ACM Transactions on Programming Languages and Systems, 15(5): 745–770, 1993.
25. M. Herlihy and J. Wing. “Linearizability: a Correctness Condition for Concurrent Objects.” ACM Transactions on Programming Languages and Systems, 12(3): 463–492, 1990.
26. H. Sundell and P. Tsigas. NOBLE: A Non-Blocking Inter-Process Communication Library. In Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers, 2002.
27. P. Tsigas and Y. Zhang. Integrating Non-blocking Synchronization in Parallel Applications: Performance Advantages and Methodologies. In Proceedings of the 3rd ACM Workshop on Software and Performance, pages 55–67. ACM Press, 2002.
28. M. Herlihy. “A methodology for implementing highly concurrent data objects.” ACM Transactions on Programming Languages and Systems, 15(5): 745–770, November 1993.
29. J. Mellor-Crummey and M. Scott. “Algorithms for scalable synchronization on shared memory multiprocessors.” ACM Transactions on Computer Systems, 9(1):21–65, 1991.
30. J. Turek, D. Shasha, and S. Prakash. “Locking without Blocking: Making Lock Based concurrent Data Structure Algorithms Nonblocking.” In Proceedings of the 11th ACM SIGACT-SIGMOD-SIGARTSymposium on Principles of Database Systems, pages 212–222, 1992
31. W. Pugh. “Concurrent Maintenance of Skip Lists.” Technical Report, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, CS-TR-2222.1, 1989.
32. W. Pugh. Skip Lists: “A Probabilistic Alternative to Balanced Trees.” In Communications of the ACM, 33(6):668{676, June 1990.