Parallel Data Structures Library: Queues and Linked Lists

Daniel Rivera Ruiz **Fatima Mushtaq**

Courant Institute of Mathematical Sciences
New York University
New York, NY 10021
{drr342,fm1529}@nyu.edu

Abstract

We've provided concurrent implementation for two data structures: Queue and Linked List in OpenMP (a shared-memory architecture). Like other programming languages, the purpose of the paper is primarily to support and test concurrent thread-safe data-structures for OpenMP. We've provided two implementations for each data-structure: lock-based and lock-free implementation. Both approaches are thoroughly tested for correctness and scalability. An efficient implementation of lock-free data-structures is developed using C++ atomic library for Linked Lists and using OpenMP Atomic directive for Queue Implementation. Both the data-structures scale well as the problem size and the number of threads increase. Lock-free implementation has proven to be the most efficient for both data-structures.

1 Introduction

A concurrent data structure is one that stores and organizes data for access by multiple threads (or processes) on a computer. In this project, we focus our attention to concurrent data structures in a shared-memory environment where multiple threads are concurrently reading and/or writing. An assumption for our discussion will be that all concurrent data structures under consideration are linearizable. A list of events is said to be linearizable if and only if its invocations were serializable, but some of the responses of the serial schedule have yet to return.
When building a concurrent data structure, there are four techniques most commonly used:

1. **Coarse-Grained Locking.** This is the naivest approach, in which a big lock wraps all the data at once. Several large open source projects have relied on course-grained locking. The Linux kernel used a big kernel lock when it first introduced support for simultaneous multithreading in Linux 2.0. Work began in 2008 to remove the big kernel lock, and it was carefully replaced with fine-grained locks until it was removed in Linux 2.6.39.
2. **Fine-Grained Locking.** This method introduces multiple locks, each of which is responsible for protecting a region of the data. Operations on the data are responsible for acquiring one or more of these locks in order to read, modify, and/or write. Fine-grained locking can improve the overall throughput of a concurrent system. However, one must be careful to avoid the perils of concurrency including deadlock, livelock, starvation, preemption, priority inversion, convoying, etc.
3. **Lock-Free Programming.** A lockfree implementation of a concurrent data structure is one that guarantees that some thread can complete an operation in a finite number of steps regardless of the execution of the other threads. Lockfree programming uses atomic operations, such as test-and-set, compare-and-swap, etc., to maintain a consistent state. Lockfree

programming can improve system throughput and has desirable liveness properties. However, it is also tricky to design and implement correctly.

4. **Transactional Memory.** This method allows several memory operations to be grouped together and execute atomically. Either the entire sequence of operations appears to occur atomically, or else the system state is unchanged. Transactions are a very familiar concept in a database management system. Transactional memory is still somewhat of an exotic concept. It can be implemented either in software or in hardware, or as a hybrid in software with hardware support.

In this project, we've focused on the fine-grained lock and lock-free implementations of two data structures: linked lists and queues in OpenMP with C++ Libraries. We've provided a comparison between the two implementations for each API of both the data-structures in terms of execution time and speed-up as the problem size scales.

2 Related Work

2.1 Queue

The first lock-free implementation of Queue data-structure of Michael and Scott [1] is the most effective and practical dynamic memory allocation algorithm in the literature. Their algorithm introduces a sentinel node which allows it to change the head and the tail pointer separately. It uses a compare-and-swap (CAS) operation in place of locks. The *enqueue* method only updates the tail pointer and the *dequeue* method only writes to the head pointer, however, *dequeue* reads both head and tail pointers to check if the queue is empty. The *enqueue* method uses two CAS statements to fully push a new node to the queue. The Michael and Scott algorithm sets either the last or second to last element as the tail and tail is then fixed while dequeuing an element. This happens as there are two CAS statements and they both have to succeed for an enqueue to the queue, therefore, multiple attempts in order to succeed the CAS statements may lead to a broken queue case therefore marking the last two pointers as tail is necessary in some cases.

Edya et al.[2] proposed an optimistic lock-free approach to the Michael-Scott Lock-free Queue Algorithm. They replaced the need for two CAS statements in both *enqueue* and *dequeue* methods by one: using a doubly linked list and by reversing the direction in which the elements are enqueued. This guaranteed that the list is never broken and the nodes are enqueued in the right order. The key idea in their algorithm is to logically reverse the direction of enqueues and dequeues to/from the list. If enqueues were to add elements at the beginning of the list, they would require only a single CAS, since one could first direct the new nodes next pointer to the node at the beginning of the list using only a store operation, and then CAS the tail pointer to the new node to complete the insertion. However, this re-direction leaves with a problem at the end of the list: dequeues would not be able to traverse the list backwards to perform a linked- list removal. If a previous pointer is found to be inconsistent, they ran a fixList method along the chain of next pointers which is guaranteed to be consistent. Since, previous pointers become inconsistent as a result of long delays, not as a result of contention, the frequency of calls to fixList is low. The result is a FIFO queue.

[3] Aldinucci et al. studied the problem of how to use efficient synchronization mechanisms for implementing fine grained parallel programs on modern shared cache multi-core architectures. They studied the Single-Producer/Single-Consumer (SPSC) coordination using unbounded queues and proposed a novel unbounded SPSC algorithm capable of reducing the row synchronization latency and speeding up Producer-Consumer coordination is presented.

2.2 Linked List

A linked list is a collection of nodes. Each node is composed of a value, and a possibly null link reference to another node. One of the basic operations a concurrent linked list must support is insertion, which locates the predecessor node where the new node is to be inserted and attempts a compare-and-swap (CAS) operation on the link reference on the predecessor node. The biggest challenge, however, presents itself in the form of the deletion operation, which attempts to remove a node from the linked list.

Several approaches have been developed over time to handle deletion in a lock-free concurrent linked list. In 1990, Valois [4] introduced the first lock-free linked list algorithm. Consistency is maintained by using auxiliary nodes which are defined as nodes that do not store values. Every normal node in the list is required to have an auxiliary node as its predecessor and its successor. With this design, consistency is maintained at the cost of a two-fold storage overhead. The modern lock-free linked list design was introduced by Harris in 2001 [5]. In this design two CAS operations are used to perform deletion. The first CAS operation marks the node as logically deleted while the second one physically removes the node. The linearization point is in the first CAS operation. This separation of concerns between logical deletion and physical deletion is a powerful technique and it forms the basis of many modern lock-free data structures.

3 Implementation

3.1 Queue

There are two implementations of the concurrent Queue data-structure that are developed in OpenMP. One is based on the fine-grained locking. It uses the OpenMP Locks (test, set, unset) statements. The other is the Lock-Free implementation which uses the atomic directive of OpenMP as a replacement of compare-and-swap (CAS) statement. Unfortunately, CAS operations are not inexpensive since they might fail to swap operands when executed and may be re-executed many times, thus introducing other sources of potential overhead, especially under high contention [6]. Considering the overhead of multiple tries for a successful CAS when several threads (or processes) try to *enqueue* or *dequeue* at the same time, our lock-free queue implementation uses only one atomic capture directive to avoid retries. This is achieved by using array as an underlying data-structure. There are two separate index pointers to access the head and tail. Like the Michael and Scott implementation [1], the *enqueue* method only changes the tail pointer and the *dequeue* only changes the head pointer and both change it atomically.

```
bool enqueue(T x){
bool enqueued = false;

if( !isFull() ){
   int currentTail = tail;

   #pragma omp atomic capture
   {
      currentTail = tail;
      tail += 1;
   }
   queue[ currentTail ] = x;
   enqueued = true;
}

return enqueued;
}
```

```
T dequeue(){
T val = -1;

if( !isEmpty() ) {
   int currentHead = head;

#pragma omp atomic capture
{
      currentHead = head;
      head += 1;
}

   val = queue[ currentHead ];
}

return val;
}
```

Listing 1: `enqueue` and `dequeue` Lock-free Implementation using OpenMP 3.1 atomic directive.

The key idea behind the implementation is that the queue is initialized with a size that is passed to the underlying array. If the caller program does not give a size it initializes it to a default size. Because of this, for both the *enqueue* and *dequeue* operations, it is only left to move the index for array correctly, as the size stays the same. This also eliminates the ABA problem as the element is not deleted from the array when dequeued, only updated again when an enqueue happens. The atomic capture directive from OpenMP 3.1 first saves the value of a memory location and then updates its value all atomically. Therefore, for *enqueue* method, the atomic capture first saves the current tail index and then increments it by one, both steps are executed atomically, therefore no other thread or process executing in parallel can have the same index value to insert at. As each thread will have a

unique index to access, we can change the location pointed by the index in the array even without a lock or critical section. Similarly, the *dequeue* method atomically finds its index to *dequeue* from and then return the element at the index from the underlying array. The implementation of *enqueue* and *dequeue* method is provided in Listing 1 above.

3.2 Linked List

Both our implementations of the concurrent linked list (fine-grained lock and lock-free) are based on the architecture proposed by Harris [5]. Compared to the original design, the main contributions of our proposal are the following:

1. Unlike Harris', our linked list supports random insertions and deletions, since the nodes aren't necessarily sorted according to the natural ordering of the values they contain.
2. The functions supported by our linked list include not only insertion, deletion and searching, but also updating and retrieving nodes and retrieving the size of the list.

In the case of the lock-free implementation, we follow a similar structure to the one introduced by Harris, where two CAS operations are required to implement deletion. The first CAS operation is performed only by the delete function, and it effectively marks a node as logically deleted. The second CAS operation can be executed in any function that modifies the structure of the list and is the one responsible for physically deleting a marked node.

If we were to implement deletion using only one CAS operation, we could end up in a situation like the one depicted in figure 1 b), where an additional node is inserted (and lost) at the same time node 10 is deleted. By using two CAS operations, the two-step deletion of node 10 will look something like figure 2. In this case, a thread trying to perform an insertion will find that node 10 is marked, therefore postponing the insertion until having physically deleted it.

a) b)

Figure 1: Example of the deletion problem using only one CAS operation. In a) the deletion succeeds because there are no concurrent operations, but in b) the concurrent insertion of node 20 fails.

Figure 2: Two-step deletion using two CAS operations: one for logical deletion or marking of a node (left) and one for physical deletion of the node (right).

The fine-grained lock implementation of our linked list follows the same architecture, but instead of using compare-and-swap atomic operations, it introduces a lock structure that wraps each node in the list. These locks are acquired and released by the concurrent threads in a consistent manner that emulates the original CAS implementation.

In terms of the libraries utilized, the lock-free implementation relies on the standard `C++ <atomic>` library, whereas the fin-grain lock implementation is based on the `OpenMP` lock routines. Both implementations are generic and support the same set of APIs, which is summarized in table 1.

4 Experimental Setup

The experiments for both the concurrent data structures were tested on the `crunchy5` server at NYU. This server consists of 64 logical cores distributed among 4 *AMD Opteron 6272* processors, each

Table 1: Set of APIs supported by our concurrent linked list implementation. All `remove` and `set` operations return a pointer to the element that was in the list before the operation was executed.

Function	Description
`bool add (int index, const T& value)`	Insert *value* at position *index*
`bool addFirst (const T& value)`	Insert *value* at head of list
`bool addLast (const T& value)`	Insert *value* at tail of list
`void clear ()`	Remove all nodes from list
`bool contains (const T& value) const`	Search for *value* in list
`T* get (int index) const`	Retrieve pointer to element at position *index*
`T* getFirst () const`	Retrieve pointer to element at head of list
`T* getLast () const`	Retrieve pointer to element at tail of list
`int indexOf (const T& value) const`	Retrieve position of first occurrence of *value*
`bool isEmpty () const`	Check if list is empty
`void print () const`	Print contents of list
`T* remove (int index)`	Delete element at position *index*
`T* removeFirst ()`	Delete element at head of list
`T* removeLast ()`	Delete element at tail of list
`bool removeValue (const T& value)`	Delete first occurrence of *value* in list
`T* set (int index, const T& value)`	Set element at position *index* to *value*
`T* setFirst (const T& value)`	Set element at head of list to *value*
`T* setLast (const T& value)`	Set element at tail of list to *value*
`int size () const`	Retrieve size of list

one with 8 physical cores enabled for hyperthreading. Since our application relies solely on `OpenMP` and `C++` standard libraries (there is no support for distributed memory system such as `MPI`), the code can only be parallelized within a single processor, i.e. the number of logical cores available for the application is 16. To compile our code, we use `gcc-8.2` with the `-fopenmp` flag to support the `OpenMP` API.

4.1 Queue

In order to test the concurrent Lock-based as well as Lock-free Queue implementation, there are primarily two test-cases written:

TestCase-1: All threads first only *enqueue* in parallel and after all threads *dequeue* concurrently from the Queue. A running sum is calculated for both enqueued elements and then when those are dequeued. The sum must match as equal iterations are given to both *enqueue* and *dequeue*, which are equal to the Queue Size.

TestCase-2: It runs both *enqueue* and *dequeue* concurrently in random order with roughly 50% probability of each operation. That is to test the Queue is growing and shrinking concurrently and is still consistent. It calculates a running sum of enqueued and dequeued elements as well. But, for this Test Case the two sums not necessarily be equal as there could be still elements left to *dequeue* because it may be that a smaller number of iterations are given to the *dequeue* threads (as it is done randomly). However, if all of the remining elements are dequeued, the two sum of enqueued and dequeued elements must match for correctness.

1. The driver program expects two programs arguments: queue size (n) and number of threads (t). The number of enqueues or dequeue iterations is based on the queue size.
2. The n repetitions of each operation are distributed among t threads using `#pragma omp parallel for`.
3. The time elapsed for the n repetitions of each operation is then measured using `omp_get_wtime`.

4.2 Linked List

The test setup for Linked List Data-structure APIs is as follows:

1. Firstly, the selection for the kind of list (fine-grained lock or lock-free) is made by passing the appropriate argument to the experiment driver. To achieve this kind of run-time polymorphism, both implementations inherit all their methods from a common ancestor `LinkedListInterface`.
2. The experiment driver takes two additional command-line arguments: number of threads (t) and number of repetitions (n). The number of repetitions refers to how many times each operation will be performed on the linked list.
3. We initialize the list with $100,000$ random integers.
4. To perform n repetitions of each operation that the list supports, except for `size`, `isEmpty`, `clear` and `print` methods, the n repetitions among t threads are distributed using `#pragma omp parallel for`.
5. Random integers are used as values for the operations that take arguments, i.e. indices and/or values.
6. We measure the time elapsed for the n repetitions of each operation using `omp_get_wtime`.
7. Lastly, we perform a random experiment where instead of performing the same operation n times, we randomly select operations until n operations have been executed. This experiment tries to emulate a real scenario where several threads will be performing different operations on the list.

5 Evaluation and Results

5.1 Queue

For evaluation and results purposes, another implementation of concurrent queue in C++ using Pthreads and mutex is written. Each of the concurrent queue implementation: Pthreads Concurrent Queue, OpenMP Lock-based Queue and OpenMP Lock-free Queue, up to a 10,000 *enqueue* and *dequeue* operations are tested with as less than as 10 threads to as many as the number of operations. All concurrent queue implementations are consistent in results in terms of correctness. The three implementations are then compared in terms of execution time for both the *enqueue* and *dequeue* methods.

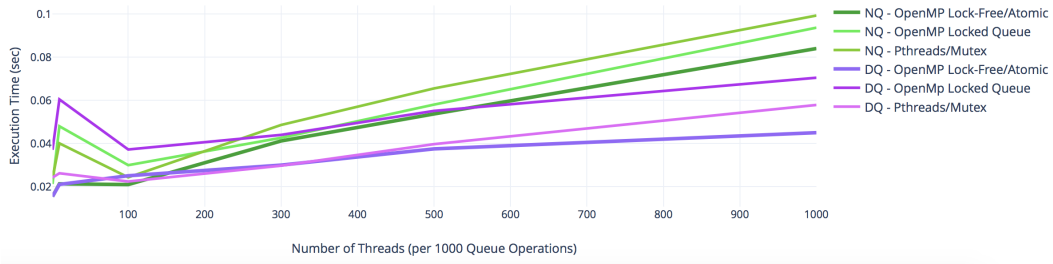

Figure 3: concurrent enqueue and dequeue execution times for C++ Pthreads/Mutex, OpenMP Lock-based and OpenMP Atomic Lock-free Implementations

The results shown in Figure 3 are taken from TestCase-1 where equal number of iterations to both *enqueue* and *dequeue* operations are given. It can be clearly seen that lock-free implementation using OpenMp Atomic directive takes the least execution time and scales with number of threads for both *enqueue* and *dequeue* methods. In general, *dequeue* in terms of execution time takes lesser time for all three implementations than *enqueue* method. Comparing the lock-based approaches, pthreads mutex-lock implementation for both *enqueue* and *dequeue* takes lesser time than OpenMP Locks. This can be explained as Pthreads library of C++ is at the layer closer to the underlying system organization, therefore pthreads-mutex lock based concurrent queue is faster than OpenMP lock-based concurrent queue.

The results shown in Figure 4 compare the speed-up for OpenMP Lock-Free Atomic Queue implementation vs the OpenMP Locked Queue implementation. Clearly, the lock-free implementation as

Figure 4: OpenMP Lock-based and OpenMP Atomic Lock-free Overall Speed-Up

it has no locks, and there's only atomic instruction to execute, we can see that it performs and scales significantly better than the OpenMP locked-queue.

5.2 Linked List

Tables 2 and 3 show the results of our experiments for both implementations of the concurrent linked list. We set the number of repetitions n for each operation to $1,000$ and ran the experiment with $t = \{1, 2, 4, 8, 16, 32, 64\}$ threads. Figure 5 shows a summary of the average results for both implementations.

Looking at the results from tables 2 and 3 and from Figure 5, we can draw the following conclusions regarding our concurrent linked lists:

- For both implementations we observe a consistent speedup as the number of threads increases, until it stabilizes at 32 threads. This result is somewhat unexpected because there are only 16 hardware threads available in the system, so we would have expected the speedup to stabilize at this number. Possible reasons to explain this behavior are thread scheduling optimization (implemented in the OS) or thread context switching optimization (implemented on hardware).
- For both implementations, the operations that access the head of the list (`addFirst`, `getFirst`, `removeFirst`, `setFirst`) are the fastest. It is reasonable to expect that these operations execute quite fast because they only access the first node in the list, thus visiting other nodes (traversing the list) is never necessary. However, these are also the only operations that do not benefit from using more threads, and in fact there is a considerable slowdown as the number of threads increases. This result is also reasonable and is closely related to the previous one: since all operations are trying to access the same location in memory (the head of the list), the work executed by all the threads is serialized under the hood to ensure correctness and consistency. Therefore, by using more threads we are only incurring into the overhead of creating them without any real benefit since all the work will be performed sequentially anyway.
- Surprisingly, the Fine-grained Lock implementation is about 5 times faster than the Lock-Free one. The most likely explanation for this result is that under the hood the `<atomic>` library is much heavier than the lock routines provided by `OpenMP`. In fact, the `<atomic>` library is built on top of locks and/or mutexes, and even if they are designed to be as hardware friendly as possible, they still introduce certain level of overhead. On the other hand, `OpenMP` provides the bare locks without any wrappers around them, which can potentially make them faster if used properly.

6 Conclusion

The concurrent data-structures are implemented in `OpenMP` using the fine-grained locking with `OpenMP` locks (test, set and unset statements) and the Lock-free approach, that for concurrent Linked List uses the C++ atomic library and for concurrent Lock-free Queue implementation uses only one atomic directive.

According to our results, the lock-free implementation of the Queue is more efficient and faster than the locked-implementation. This can be explained because as the problem size and number of threads increase, using locks adds contention and yields poor performance. It is also remarkable that we successfully implemented both *enqueue* and *dequeue* methods using only one atomic capture directive. This technique yields significant speed-up in comparison to both pthreads concurrent locked queue and `OpenMP` concurrent locked queue.
In the case of the Linked List, however, the fine-grained lock implementation using `OpenMP` turns out to be faster than the lock-free implementation using `C++ <atomic>`. Our intuition behind this is that under the hood, the thread-safe mechanisms implemented by `<atomic>` are much heavier than those implemented by `OpenMP`locks.

Table 2: Execution times (in seconds) for $1,000$ repetitions of each operation on the Fine-grained Lock implementation of the concurrent linked list.

	Threads						
Operation	1	2	4	8	16	32	64
`add`	0.5344	0.2727	0.1450	0.0802	0.0413	**0.0275**	0.0353
`addFirst`	**0.0002**	0.0007	0.0017	0.0019	0.0020	0.0023	0.0024
`addLast`	1.2138	0.6106	0.3049	0.1546	0.0790	**0.0454**	0.0546
`contains`	0.3650	0.1847	0.0943	0.0483	0.0254	**0.0160**	0.0186
`get`	0.4249	0.2165	0.1100	0.0584	0.0314	**0.0176**	0.0190
`getFirst`	**0.0001**	0.0005	0.0009	0.0012	0.0012	0.0015	0.0017
`getLast`	0.8241	0.4132	0.2071	0.1116	0.0531	**0.0295**	0.0362
`indexOf`	0.3600	0.1844	0.0987	0.0507	0.0322	**0.0286**	0.0310
`remove`	0.5198	0.2616	0.1375	0.0729	0.0390	**0.0247**	0.0252
`removeFirst`	**0.0002**	0.0010	0.0015	0.0019	0.0030	0.0054	0.0068
`removeLast`	1.1663	0.5795	0.2965	0.1496	0.0799	0.0467	**0.0463**
`removeValue`	0.4452	0.2351	0.1207	0.0640	0.0344	**0.0216**	0.0235
`set`	0.5434	0.2735	0.1388	0.0757	0.0402	**0.0228**	0.0234
`setFirst`	**0.0001**	0.0004	0.0011	0.0017	0.0017	0.0020	0.0021
`setLast`	1.1748	0.5892	0.2932	0.1481	0.0742	**0.0400**	0.0506
Random	0.5242	0.2663	0.1340	0.0755	0.0419	0.0266	**0.0245**
Average	0.5060	0.2556	0.1304	0.0685	0.0362	**0.0224**	0.0251

Table 3: Execution times (in seconds) for $1,000$ repetitions of each operation on the Lock-free implementation of the concurrent linked list.

	Threads						
Operation	1	2	4	8	16	32	64
`add`	2.1466	1.0783	0.5797	0.2957	0.1538	**0.0863**	0.0962
`addFirst`	**0.0003**	0.0008	0.0017	0.0021	0.0022	0.0023	0.0029
`addLast`	5.9411	2.9579	1.5034	0.7945	0.3900	0.2179	**0.2116**
`contains`	2.6930	1.2936	0.6919	0.3429	0.1847	0.1088	**0.1031**
`get`	2.5736	1.2853	0.6715	0.3525	0.1818	0.0974	**0.0969**
`getFirst`	**0.0001**	0.0005	0.0009	0.0016	0.0016	0.0018	0.0022
`getLast`	4.9291	2.4554	1.2302	0.6159	0.3117	**0.1602**	0.1797
`indexOf`	2.6762	1.3511	0.6768	0.3504	0.1915	**0.1082**	0.1131
`remove`	2.0450	1.0296	0.5231	0.2799	0.1470	0.0825	**0.0822**
`removeFirst`	**0.0003**	0.0010	0.0055	0.0062	0.0045	0.0067	0.0079
`removeLast`	5.7793	2.8767	1.4683	0.7614	0.3889	0.2112	**0.2103**
`removeValue`	2.4408	1.2554	0.6269	0.3331	0.1811	**0.0971**	0.0988
`set`	2.1432	1.0916	0.5642	0.2841	0.1546	0.0894	**0.0877**
`setFirst`	**0.0002**	0.0006	0.0012	0.0019	0.0020	0.0022	0.0024
`setLast`	5.7369	2.9258	1.4639	0.7324	0.3709	**0.1992**	0.2084
Random	2.6681	1.3369	0.7101	0.3678	0.2102	0.1118	**0.1024**
Average	2.6109	1.3088	0.6699	0.3451	0.1798	**0.0989**	0.1004

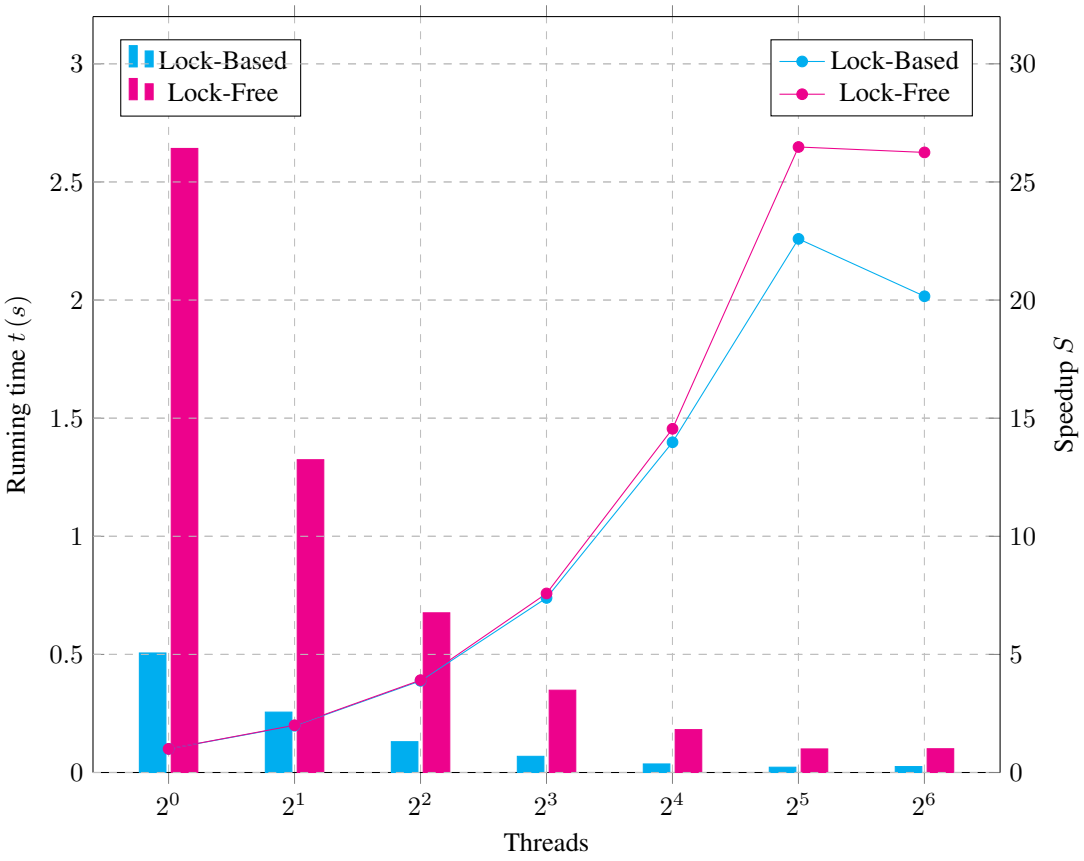

Figure 5: Summary of results for the concurrent linked list. Fine-grained lock (Lock-Based) implementation is shown in cyan and Lock-Free implementation shown in magenta. On the left vertical axis, we plot the average running time for each implementation as a bar chart. On the right vertical axis, we plot speedup of the average running time with respect to the 1-thread execution as a line chart.

References

[1] M. L. S. Maged M. Michael, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," 1996.

[2] E. Ladan-Mozes and N. Shavit, "An optimistic aapproach to lock-free fifo queues," 2004.

[3] P. K. M. M. M. Aldinucci, M. Danelutto and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," 2012.

[4] J. D. Valois, "Lock-free linked lists using compare-and-swap," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, (New York, NY, USA), pp. 214–222, ACM, 1995.

[5] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, (London, UK, UK), pp. 300–314, Springer-Verlag, 2001.

[6] G. E. K. R. L. K. G. G. Orozco, D.A., "Toward high- throughput algorithms on many-core architectures," 2012.

[7] M. Moir and N. Shavit, "Concurrent data structures," in *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*, pp. 47–14 47–30, 2004. Chapman and Hall/CRC Press.