Practical Blocking and Non-Blocking Parallel Linked List and Queue Algorithms

Iju Lee
NYU Courant CS
Taipei, Taiwan
ijl245@nyu.edu

Shih-Yao Chou
NYU Courant CS
Taipei, Taiwan
syc574@nyu.edu

Jiyuan Lu
NYU Courant CS
Nanjing, China
jl11046@nyu.edu

Abstract

Recent years, running parallel programs on multicore systems has become a common fashion. As programmers, we can benefit from parallel data structures when building high performance and scalable programs on multicore systems. This motivation drove us to survey parallel data structures for multicore systems. Although we found plenty of implementations based on blocking and non-blocking techniques, most of them were originally designed, tested, and employed on multiprogramming systems. Thus, we were curious about whether these parallel data structures designed for multiprogramming systems worked as well for popular multicore systems. In this paper, we first implemented a lock-based blocking linked list and a lock-based blocking queue. Then we designed a lock-free non-blocking linked list and a lock-free non-blocking queue based on the atomic primitive compare_and_swap (CAS). Finally we proved the scalability of our lock-free data structures and compared their performance and with their lock-based counterparts using a microbenchmark. We found out that non-blocking parallel structures consistently outperformed their blocking counterparts in terms of both performance and scalability. Experiments showed that our non-blocking data structures are both fast and efficient. In the end, we gave recommendations on the appropriate data structures to use when running programs in a multicore system.

Keywords: multicore programming, non-blocking, lock-free linked-list, lock-free queue, compare-and-swap

1 Introduction

Parallel data structures are mainly categorized into blocking and non-blocking designs. The blocking concept involves several locking techniques, such as Mutex, Spinlock. Mutexes are best used in user space programs where sleeping of processes does not affect the system in a catastrophic way. Spinlock is efficient if threads are likely to be blocked for only short periods, and can be more efficient than

calling heavyweight thread synchronization functions. The non-blocking concept mainly involves lock-free and wait-free data structures, both of which can guarantee thread progress but to a different extent. Besides, wait-free algorithms are mostly less efficient than lock-free algorithms. And a wait-free data structure is usually based on an underlying lock-free data structure. Few studies talked about wait-free data structures. The reason might be that the concept of wait-free is complex and is thus more difficult to implement. Our focus is on lock free data structures.

In the 1980s, distributed memory and parallel computing developed, as did a number of concurrent data structures for writing programs under such a multiprogrammed environment. As we approached the Millennium, SMT (Simultaneous MultiThreading) architectures became more and more popular. Multicore processors represent an evolutionary change in conventional computing and set the new trend for high performance computing. This trend gives us much hope about enhancing performance and scalability of existing systems beyond the traditional multiprogramming techniques. As a result, we expect to see an emerging need for a parallel data structure library suitable for multicore systems. Luckily, many parallel data structures originally designed for multiprogramming systems can also be used for multicore systems, requiring few or even no modifications. In this paper, we implemented both blocking and non-blocking versions of parallel linked lists and queues, and inspected their performance and scalability respectively under different test environments. Finally, we gave readers recommendations on which data structures are more suitable under different scenarios in a multicore system. Our non-blocking linked list and queue implementations served as a first step towards building such a robust, efficient, and scalable parallel data structure library.

2 Literature Review

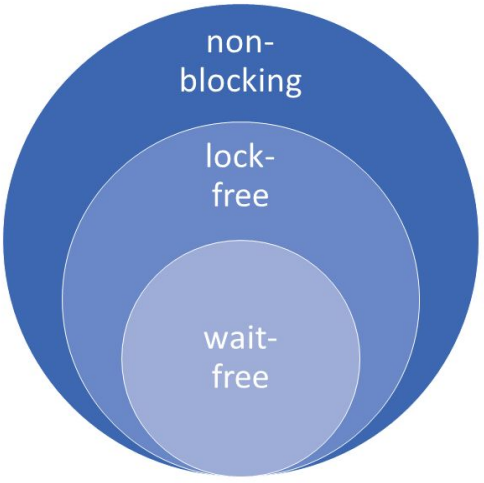


Figure 1: The concept of non-blocking, lock-free, and wait-free data structures.

Almost every common data structure has its parallelized version. To name just a few, there are stacks, queues, deques, priority queues, maps, sets, linked lists, binary search trees, graphs, skip lists, union-find, hash tables, and dynamic resizable arrays (vectors), each supporting its unique set of operations. Key criterions for evaluating a parallel data structure include performance, correctness, and possibly also scalability. Although some of the parallel data structures were originally designed for multi-programmed environments, they also work naturally for multi-threaded settings.

Since our focus was on linked lists and queues, we carried out a detailed survey on the existing parallel linked lists and parallel queues implementations, put them into taxonomy, and identified their pros and cons.

2.1 Parallel Queue Data Structures

A parallel queue can be blocking or non-blocking, and the underlying data structure can be a linked list or an array.

Blocking queues, which incorporate locking mechanisms with a sequential queue, are relatively easy to implement. Michael and Scott gave a two-lock concurrent queue [8] implementation. The two-lock queue outperforms a single lock when several processes are competing simultaneously for access, and appears to be the algorithm of choice for busy queues on machines with non-universal atomic primitives, such as test_and_set.

Most non-blocking queues are lock-free designs. Michael and Scott's non-blocking concurrent queue [8], Shann et al's non-blocking queue [20], Hoffman et al's baskets queue [16], Morrison et al's concurrent queue [11] are all lock-free designs. Michael and Scott's non-blocking queue consistently outperformed the best known alternatives at their time, and was suited for machines that provide a universal atomic primitive, such as compare_and_swap. Shann et al's non-blocking queue solved the memory management problem for many link-list-based queues that a dequeued node could not be freed or reused without proper handling. Hoffman et al's baskets queue was built upon Michael and Scott's queue and addressed the problem in many concurrent dynamic-memory lock-free queues that the data structure offered no more parallelism than that provided by allowing concurrent accesses to the head and tail, by introducing a new form of parallelism among enqueue operations that created baskets of mixed-order items instead of the standard totally ordered list. Morrison et al's concurrent queue relied on fetch_and_add, a less powerful synchronization primitive than compare_and_swap, and outperformed combining-based implementations.

People have also designed wait-free non-blocking queues. Yang et al [21] proposed a wait-free queue based on fetch_and_add, which in principle performed better than designs based on compare_and_swap under high contention. Matei David [22] proposed a wait-free queue for a specific scenario where only one process performed enqueue operations and any number of processes performed dequeue operations. Kogan et al [23] proposed a wait-free queue built upon Michael & Scott's lock-free queue using a priority-based helping scheme in which faster threads helped the slower peers to complete their pending operations.

For the underlying data structure: Michael and Scott's, Hoffman et al's, Morrison et al's, and Kogan et al's implementations are all based on linked lists. Shann et al.'s and David's implementations are based on arrays. Yant et al's implementation is based on a singly-linked list of equal-sized array segments.

2.2 Parallel Linked List Data Structures

A parallel linked list can be blocking or non-blocking, singly-linked or doubly-linked.

A blocking linked list is relatively easy to implement, for example, we can use one or more mutual exclusion locks to protect the list shared among multiple threads.

Most non-blocking linked lists are implemented using the lock-free technique. Harris [4] proposed a conceptually simple lock-free singly-linked list that supported linearizable insertion and deletion operations, Sundell and Tsigas [24] proposed a lock-free doubly-linked list that supported parallelism for disjoint accesses and was suitable for systems with high concurrency and non-uniform memory architecture. Braginsky and Petrank [25] proposed a locality-conscious lock free singly-linked list that made use of consecutive chunks of memory each holding a number of entries. Fomitchev and Ruppert [26] proposed a lock-free singly-linked list that used backlinks which were set when a node was deleted so that concurrent operations visiting the deleted node could recover, and the worst-case amortized cost of the operations was linear in the length of the list plus the contention.Valois [27] proposed a lock-free singly-linked list using the compare_and_swap primitive that achieved comparable performance with its blocking counterpart using spin locks.

Built upon lock-free lists, people have also proposed many wait-free designs.Timnat et al [28] proposed a wait-free singly-linked list based on the compare_and_swap primitive and the fast-path-slow-path methodology which achieved competitive performance with that of Harri's lock-free list, while providing the wait-free guarantee required for real-time systems. Kogan et al [29] introduced the fast-path-slow-path methodology for creating efficient wait-free algorithms, where the data structure

executed the efficient lock-free version most of the time and reverted to the wait-free version only when things go wrong. They showed that the performance of a wait-free linked-list ccould be dramatically improved with this proposed methodology. Afek et al [30] proposed a wait-free list that combined the advantages of both lock-free and wait-free technique, i.e., fast when contention was low and guaranteed progress of any individual operation. However, they only proved the performance gain theoretically but did not conduct any experiments to support their claim.

2.3 Comparing lock-based, lock-free, and wait-free data structures

Lock-based data structures are data structures that use mutexes and locks to protect the shared data. We can either use one mutex to protect the entire data structure, or more than one to protect different smaller parts of the data structure in order to allow greater levels of parallelism. Lock-based data structures are simple to implement in that we can easily design a lock-based data structure by adding mutexes and locks based on a sequential one. However, one major drawback of such lock-based data structures is that they provide very limited parallelism among threads.

Lock-free data structures, on the other hand, do not use mutexes. Instead, “locks” with a finer granularity are employed, namely atomic primitives such as compare_and_swap or test_and_set. Lock-free data structures have the advantage that they allow more than one thread to access the data structure concurrently. Also, if one of the threads accessing the data structure is suspended by the scheduler, other threads will still be able to complete their operations without waiting for the suspended thread.This enables some threads to make progress at every time step. Theoretically speaking, lock-free data structures could have better performance than their lock-based counterparts because they create the opportunity for more parallelism. However, in reality whether this is true depends on implementations. And rigorous experiments are needed when one wants to compare the performance of a lock-free data structure with a lock-based one. Another thing to note is that lock-free data structures are more difficult to be designed and to be verified than lock-based data structures.

Wait-free data structures build upon lock-free data structures with the additional property that every thread accessing the data structure can complete its operation within a bounded number of steps, regardless of the behavior of other threads.This can be done using the helping mechanism where a thread is assisted by other threads when it gets into trouble. Wait-free data structures are even more difficult to implement correctly and they usually have worse performance than lock-based data structures in order to satisfy the progress guarantee. Therefore, some clever designs use the fast-path-slow-path mechanism such that a wait-free data structure executes

its lock-based version most of the time and only reverts to wait-free when needed in order to satisfy the progress guarantee.

3 Proposed Idea

According to the literature reviews, we decided to implement a lock-free linked-list and queue. On the other side, we also built a blocking counterpart for these two data structures by leveraging mutex respectively. After conducting the experiments, we analyzed the strengths and weaknesses (e.g. performance, scalability) between blocking and non-blocking of aforementioned data structures.

We discovered that most papers in the literature conducted their experiments for parallel data structures under a multiprogramming environment. However, few of them discussed the performance of their data structures using a multicore processor. Moreover, the papers focus mainly on the performance gain but lack formal experimental analysis on scalability.

3.1 Lock Free Queue

FIFO (First-In-First-Out) queue is an essential data structure. It plays an important role in many algorithms and applications. Various CPU scheduling algorithms are implemented using the queue data structure. Algorithms like BFS also use a queue to store the nodes. Therefore, a simple yet effective non-blocking queue implementation has the possibility for enhancing the program performance significantly. In this paper, we first build a blocking queue by using a single mutex and use it as the benchmark. Then, we designed a lock-free queue based on the compare-and-swap (CAS) operation that is common in most modern processors.

Our lock-free queue implementation is based on Michael and Scott's blocking queue algorithm [8]. It supports two key operations: queue_push (synonym for enqueue) and queue_pop (synonym for dequeue). We also implemented two utility functions: queue_new for creating and initializing a new queue, and queue_delete to remove all elements from the queue and deallocate the memory space occupied by the nodes.

The utility functions are used to create or destroy queue objects. The queue_new function first allocates a free node, making it the only node in the linked list, and directs both Head and Tail pointers to point to it. The Head pointer always points to a dummy node, which is the first node in the queue. The Tail pointer always points to a node, either the last or second to last node, in the queue. The queue_delete function

cleans up the queue by iterating through the queue and freeing each node encountered.

The queue_push operation allocates a new node and copies the enqueued value into the new node. Then it keeps trying until the enqueue operation is done. Inside the loop, it uses a CAS operation and tries to swing the Tail pointer to the next node. The queue_pop operation uses a spinlock and keeps trying until the dequeue operation is done. Inside the loop, it tries to swing the Head pointer to the next node. It checks if the queue is empty and if so it returns value 0. Otherwise, it moves the Head pointer to the next node in the queue and frees the old dummy node, and returns the value of the head element in the queue.

```
int queue_new(queue *q) {
    // sentinel
    node *new_node = calloc(1, sizeof(node));
    if (!new_node)  {
        return -errno;
    }

    memset(q, 0, sizeof(queue));
    q->head = q->tail = new_node;

    return 0;
}
```

```
 int queue_delete(queue *q){
     if (q->tail && q->head) {
         node *curr = q->head;
         node *tmp;

         // iterate through nodes
         while (curr != q->tail) {
             tmp = curr->next;
             free(curr);
             curr = tmp;
         }

         free(q->head);
         memset(q, 0, sizeof(queue));
     }

     return 0;
 }
```

```
int queue_push(queue *q, void *val) {
    node *tail;
    node *new_node = calloc(1, sizeof(node));
    if (!new_node) {
        return -errno;
    }

    new_node->val = val;
    do {
        tail = q->tail;
        if ( CAS(&q->tail, tail, new_node)) {
            tail->next = new_node;
            break;
        }
    } while (1);
    ANF(&q->count);

    return 0;
}


void* queue_pop(queue *q) {
    void *val = 0;
    node *head;

    /* A little spinlock.
     * Try to contend for q->head.
     * After getting it, assign it to zero,
     * thid thread prevents other threads from getting it */
    do {
        head = q->head;
    } while (head == 0 || !CAS(&q->head, head, 0));

    // queue is empty
    if (head->next == 0) {
        q->head = head;
        return 0;
    }
    val = head->next->val;
    q->head = head->next;

    SNF(&q->count);
    free(head);

    return val;
}
```

Figure 2: Operations of a non-blocking parallel queue

3.2 Lock Free Linked List

Since linked lists are one of the most basic data structures used in program design, and so a simple and effective non-blocking linked-list implementation can serve as the basis for many data structures. In here, we firstly built a blocking linked-list by using mutex, and noticed that its computing performance was not as expected since the synchronization was used in the implementation of the operations/objects, which implied there was a great amount of contentions among threads and they sometimes had to wait for any slow operation among them to complete. Thus, we also implemented a lock free linked-list based on compare-and-swap (CAS) operation found on contemporary processors, and then compared the computing performance and scalability between blocking and non-blocking linked lists.

In our lock free linked list implementation, there are three operations which are Insert(), Delete(), Search(). All of them take a key as the only input and return boolean showing success or failure. The linked list here is an ordered list using key to determine the position of nodes.

Insertion is guaranteed to be atomic by using a single CAS on the next pointer of the proposed predecessor, this CAS operation can ensure that both sides of the insertion still keep the original connection at the moment of updating, which implies no any other thread has inserted new nodes in this position or deleted the original right adjacent node yet.

```
int list_insert(list *l, int val) {
    node *right_node, *left_node;
    right_node = left_node = NULL;
    node *new_node = (node*) malloc(sizeof(node));
    new_node->next = NULL;
    new_node->val = val;
    while(1) {
        right_node = list_search(l, val, &left_node);
        if ((right_node != l->tail) && (right_node->val == val)) {
            return 0;
        }
        new_node->next = right_node;
        if (CAS(&(left_node->next), right_node, new_node)) {
            return 1; }
    }
}
```

Deletion is a bit more sophisticated, it uses two separate CAS operations, first CAS is for atomically assigning the right adjacent node of the target node to the next pointer of the left adjacent node of the target node. However, this approach is still insufficient since other threads may insert some new nodes between the target node and its right node at the same time. So in the above case, we will lose those newly added nodes. Thus, we add another CAS to circumvent this difficulty. Before connecting the left and right adjacent nodes of the target node together, we need to mark the next pointer of the target node beforehand. This approach indicates the target node is logically deleted. So even though the structure of the list is temporarily retained but can still avoid incoming new nodes being inserted between a logically deleted node and its original right node.

```
int list_delete(list *l, int val) {
    node *right_node, *right_node_next, *left_node;
    right_node = right_node_next = left_node = NULL;
    while (1) {
        right_node = list_search(l, val, &left_node);
        if ((right_node == l->tail) || (right_node->val != val)) {
            return -1;
        }
        right_node_next = right_node->next;
        if (!is_marked((long) right_node_next))
            if (CAS(&(right_node->next), right_node_next,
                get_marked((long) right_node_next)))
                break;
    }
    if (!CAS(&(left_node->next), right_node, right_node_next)) {
        right_node = list_search(l, right_node->val, &left_node);
    }
    return val;
}
```

Find operation searches the target node and returns the result showing success or failure by using boolean.

```
int list_find(list *l, int val) {
    node *right_node, *left_node;
    right_node = list_search(l, val, &left_node);
    if ((right_node == l->tail) || (right_node->val != val)) {
        return 0;
    } else {
        return 1;
    }
}
```

Search method is important and used by all the aforementioned operations, it mainly helps to find the node with the target key. Moreover, this method also sets several conditions to ensure the validity of each operation in the parallelism case. Firstly, the search key needs to be greater than the key of the left node and needs to be smaller or equal to the right node. Thus, the right node will be the target node. Secondly, we examine that both the target node and its left node must be unmarked. Lastly, this method will eliminate all the marked nodes between the target node and its left node.

```
node* list_search(list *l, int val, node **left_node) {
    node *left_node_next, *right_node;
    left_node_next = right_node = NULL;
    while(1) {
        node *t = l->head;
        node *t_next = l->head->next;
        /* Find left_node and right_node */
        while (is_marked((long) t_next) || (t->val < val)) {
            if (!is_marked((long) t_next)) { // valid
                (*left_node) = t;
                left_node_next = t_next;
            }
            t = (node*) get_unmarked((long) t_next);
            if (t == l->tail) break;
            t_next = t->next;
        }
        right_node = t;

        /* Check nodes are adjacent */
        if (left_node_next == right_node){
            if (!is_marked((long) right_node->next))
                return right_node;
        }

        /* Remove one or more marked nodes */
        if (CAS(&((*left_node)->next), left_node_next, right_node)) {
            if ((right_node == l->tail) && !is_marked((long) right_node->next))
                return right_node;
        }
    }
}
```

Figure 3: Operations of a non-blocking parallel linked list

To sum up, the above algorithm uses compare-and-swap and a specific mark of each node to ensure each operation can work as expected in parallel.

4 Experimental Setup

We run our experiments on CIMS crunchy3 machine that has four AMD Opteron 6136 processors (2.4 GHz, 32 cores) with 128GB shared memory running the CentOS 7 operating system.

Similarly to the evaluation of other parallel containers [7], we have designed our experiments by generating a workload of different operations (enqueue/dequeue for queues, insert/delete/search for linked lists). In the experiments, we varied the number of threads starting from 1 and exponentially increased their number to 32. Every active thread executed a number of operations on the shared container, starting from 1,000,000, and increased exponentially to 16,000,000 for queues; starting from 5,000, and increased exponentially to 80,000 for linked lists. We measured the real time (in seconds) that all threads needed in order to complete. Each iteration of every thread executed an operation with a certain probability. Finally, we varied the distribution of the operations and recorded the performance of the corresponding parallel container.

The operations supported by our data structures are as follows:

Lock-free Queue:

1. Enqueue (*queue_push(x)*): Inserts a new element at the end of the queue.
2. Dequeue (*queue_pop(x)*): Removes the element from the head of the queue.

Lock-free Linked-lists:

1. Insert (*list_insert(list, x)*): Inserts a new element into a specific position of the ordered linked-list based on a given key.
2. Delete(*list_delete(list, x)*): Removes an element with value x in the list.
3. Search(*list_find(list, x)*): Checks if a given element is in the list.

For lock-free queue, we set three different experiment configurations as shown below:

1. 100% Enqueue
2. 75% Enqueue, 25% Dequeue
3. 50% Enqueue, 50% Dequeue

For lock-free linked-list, we set four different experiment configurations as shown below:

1. 100% Insert
2. 75% Insert, 25% Delete
3. 50% Insert, 50% Delete

4. 30% Insert, 20% Delete, 50% Search

Moreover, we also offered some extra operations of queue which we did not use in the experiment. They were mainly built for adding value and convenience for users when running programs in parallel:

1. Search (*queue_peek(q)*): Checks if an element is in the queue or not.
2. Size (*queue_size(q)*): Gets the size of the queue.

5 Experiments & Analysis

In our performance analysis, we compare the non-blocking (lock-free) approach with the blocking approach in queues and linked lists, respectively. For the blocking and non-blocking linked list, the experiment results and analysis are as follows:

A matrix of plots is shown in figure 4, where we vary the number of operations in the horizontal direction and vary the distribution of operations in the vertical direction. Four observations can be made:

1. The blocking linked list performs better when there is few or no parallelism. For example, when we execute the program sequentially. However, a non-blocking linked list is preferred when there are multiple threads in the program.

2. As we increase the number of threads, the performance of the non-blocking linked list increases, while the performance of the blocking linked list decreases. This is because a non-blocking linked list utilizes the independence among operations so that several threads can read or write different parts of the list at the same time, while in a blocking linked list, a thread locks down the whole container when it is performing an operation, and thus excludes other threads from accessing the list.

3. Notice the intersection point of the blocking and the non-blocking linked list. As the problem size increases, the point moves to the left, which means fewer numbers of threads. This indicates that as problem size increases, a non-blocking queue needs fewer threads in order for the gain from parallelism to overcome the overhead in its implementation.

4. Comparing the setting where 50% of the operations are “search”, which is non-list-modifying, to the rest configurations where all operations are list-modifying. We notice that the speed for the former is about twice compared to the latter. This is because search operations do not incur contentions and can be executed fully in parallel. This also shows that our

non-blocking linked list may not be suitable for “write intensive” scenarios where there are multiple writers but are good for “read intensive” scenarios where there is a single writer but multiple readers.

Figure 5 shows the speedup versus log-scaled number of threads for 80,000 operations. We notice that for each operation distribution setting, the non-blocking linked list appears to scale logarithmically with the number of threads.

Figure 6 shows the best speedup that can be achieved for a non-blocking linked list with different problem size and under different operation distribution settings. We can see that it is possible for the non-blocking linked list to achieve upto near 10x speedup in a multicore environment. When comparing across columns, we see that the best speedup increases as data size increases. When comparing across rows, we notice that the best speedup appears when running the operation setting that has 75% write and 25% read with an 80,000 data size.

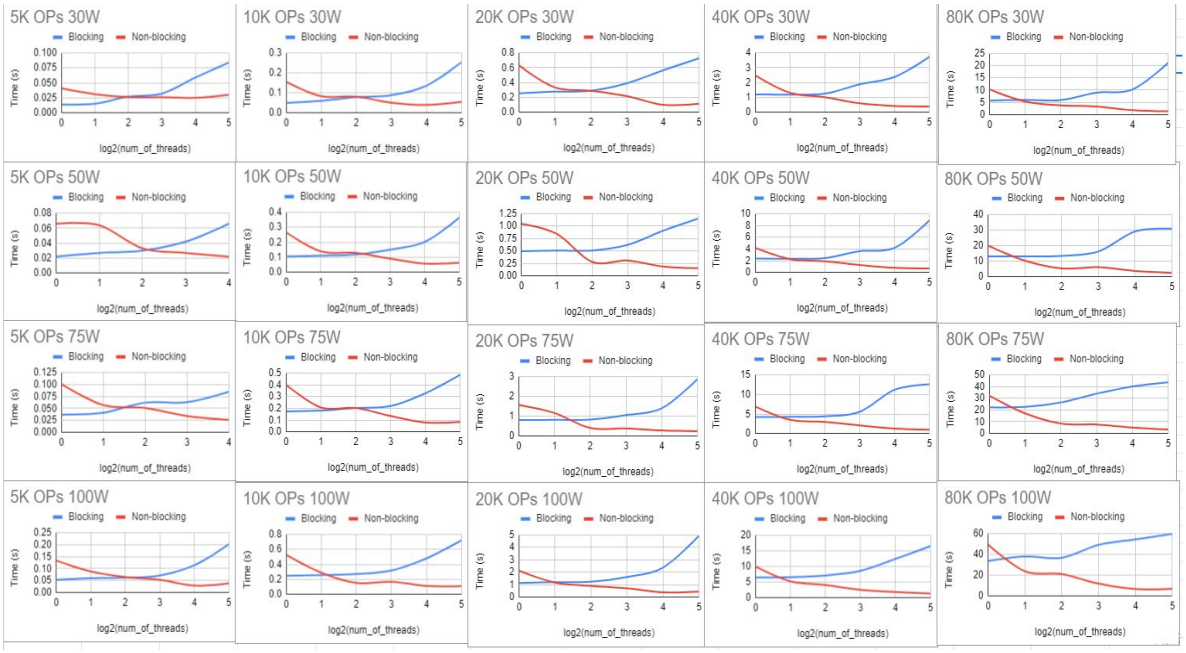


Figure 4: Time vs. log-scaled number of threads for linked lists

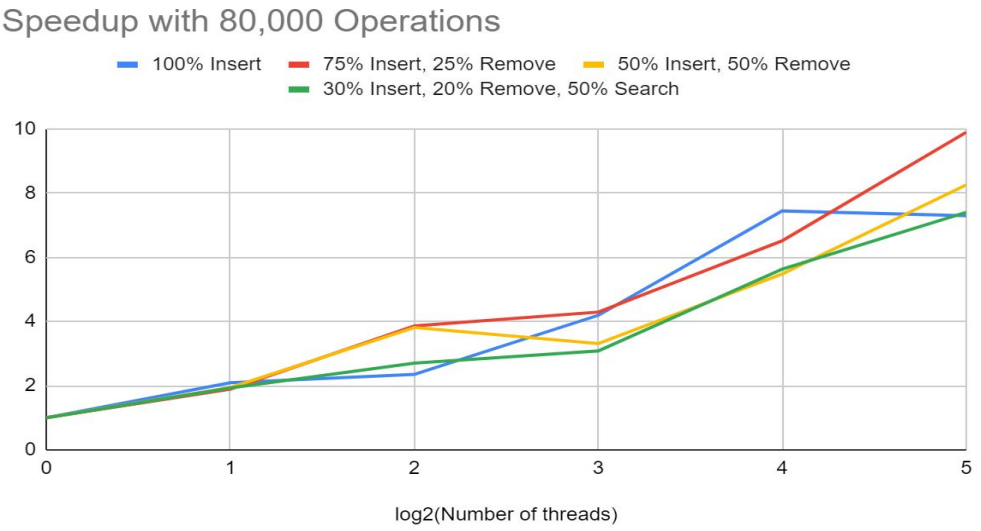


Figure 5: Speedup of the non-blocking linked list with 80,000 operations

Non-Blocking Linked List Best Speedup	5K	10K	20K	40K	80K
100% Insert	4.62 (16 threads)	4.83 (32 threads)	5.63 (16 threads)	7.55 (32 threads)	7.44 (16 threads)
75% Insert, 25% Remove	3.88 (16 threads)	5.08 (16 threads)	6.35 (32 threads)	7.08 (32 threads)	9.89 (32 threads)
50% Insert, 50% Remove	3.00 (16 threads)	4.67 (16 threads)	6.08 (32 threads)	6.15 (32 threads)	8.25 (32 threads)
30% Insert, 20% Remove, 50% Search	1.64 (16 threads)	3.97 (16 threads)	6.32 (16 threads)	6.38 (32 threads)	7.4 (32 threads)

Figure 6: Best speedup of the non-blocking linked list

When it comes to the comparison between blocking and non-blocking linked lists, the experiment results and analysis are as follows:

A matrix of plots is shown in figure 7, where we vary the number of operations in the horizontal direction and vary the distribution of operations in the vertical direction. We notice that there is not much difference among different operation distribution settings.

Figure 8 shows that a blocking queue could have better performance than a non-blocking queue when there is few or no parallelism. However, a non-blocking design is preferred when we have multiple threads in the program. Also, we notice

that as the number of threads increases, the performance decreases for both queues. The execution time increases linearly for the blocking queue but logarithmically for the non-blocking queue.

We are also surprised to see that we could not benefit much from parallelism for a non-blocking queue. Our guess is that a non-blocking queue inherently suffers from contentions among threads under a “multi-producer or multi-consumer” scenario, where multiple threads are competing for head deletions or tail insertions.

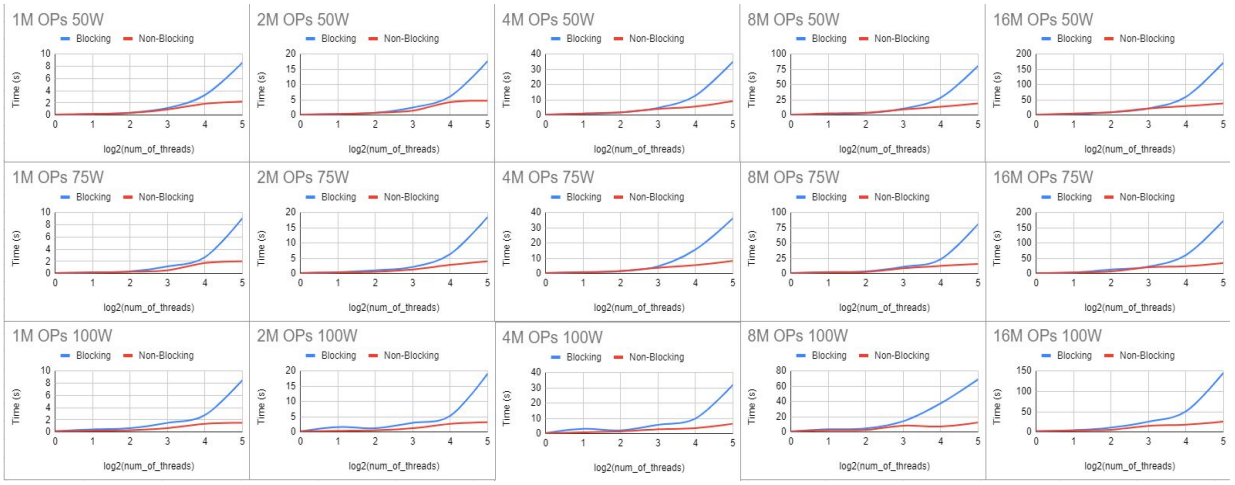


Figure 7: Time vs. log-scaled number of threads for queues

Figure 8: Performance comparison between a blocking and non-blocking queue, with 16 million operations, of which half are insert operations, and the other half are delete operations.

6 Conclusions

Queues and linked lists are ubiquitous in parallel programs. The performance and scalability of them is a matter of major concern in a multicore system. In this paper, our conclusions are as follows.

- Blocking data structures have lower overheads compared to non-blocking data structures when there is few or no parallelism.
- Some concurrent data structures designed for multiprogramming systems can be used for multicore systems with few modifications.
- Our non-blocking linked list is strongly scalable and our non-blocking queue reduces the contention overhead compared to a blocking implementation. Therefore, our non-blocking data structures are recommended for parallel programs that run in a multicore system.

7 References

[1] Shahar Timnat, “Practical Parallel Data Structures”. Israel Institute of Technology. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/2015/PHD/PHD-2015-06.pdf

[2] Fatourou, P., Kallimanis, N.D. “Highly-Efficient Wait-Free Synchronization”. Theory Comput Syst 55, 475–520 (2014). https://doi.org/10.1007/s00224-013-9491-y

[3] Y. Peng and Z. Hao, "FA-Stack: A Fast Array-Based Stack with Wait-Free Progress Guarantee," in IEEE Transactions on Parallel and Distributed Systems, vol. 29, no. 4, pp. 843-857, 1 April 2018.

[4] Timothy L. Harris, “A Pragmatic Implementation of Non-blocking Linked-Lists”, in DISC '01: Proceedings of the 15th International Conference on Distributed Computing, Pages 300–314, October 2001. https://dl.acm.org/doi/10.5555/645958.676105

[5] Maged M. Michael, “High performance dynamic lock-free hash tables and list-based sets”, in SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures. August 2002. Pages 3–82. https://doi.org/10.1145/564870.564881

[6] Ori Shalev, “Split-ordered lists: Lock-free extensible hash tables”, in Journal of the ACM, May 2006. https://doi.org/10.1145/1147954.1147958

[7] Dechev D., Pirkelbauer P., Stroustrup B. “(2006) Lock-Free Dynamically Resizable Arrays”, In: Shvartsman M.M.A.A. (eds) Principles of Distributed Systems. OPODIS 2006. Lecture Notes in Computer Science, vol 4305. Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/11945529_11

[8] Maged M. Michael & Michael L. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms”, in PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, May 1996. Pages 267–275. https://doi.org/10.1145/248052.248106

[9] Maurice P. Herlihy & J M Wing, “Linearizability: a correctness condition for concurrent objects”, in ACM Transactions on Programming Languages and Systems, July 1990. https://doi.org/10.1145/78969.78972

[10] Danny Hendler, Itai Incze, Nir Shavit, Moran Tzafrir, “Flat combining and the synchronization-parallelism tradeoff”, in SPAA '10: Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures, June 2010. Pages 355–364. https://doi.org/10.1145/1810479.1810540

[11] Adam Morrison, Yehuda Afek, “Fast concurrent queues for x86 processors”, in ACM SIGPLAN Notices, February 2013. https://doi.org/10.1145/2517327.2442527

[12] Thomas R.W. Scogland, Wu-chun Feng. “ Design and Evaluation of Scalable Concurrent Queues for Many-Core Architectures”. In ICPE '15: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, January 2015. Pages 63–74. https://doi.org/10.1145/2668930.2688048

[13] Philippas Tsigas, Yi Zhang, “A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems”, in SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, July 2001. Pages 134–143. https://doi.org/10.1145/378580.378611

[14] Mark Moir, Nir Shavit, “Concurrent Data Structures”, in Handbook of Data Structures and Applications. 2004. https://www.semanticscholar.org/paper/Concurrent-Data-Structures-Moir-Shavit/5d24eee276c2dc32f0d426edbe078a045bc93639

[15] Shafiei N. (2008) “Non-blocking Array-Based Algorithms for Stacks and Queues”. In: Garg V., Wattenhofer R., Kothapalli K. (eds) Distributed Computing and Networking. ICDCN 2009. Lecture Notes in Computer Science, vol 5408. Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-92295-7_10

[16] Hoffman M., Shalev O., Shavit N. (2007) “The Baskets Queue”. In: Tovar E., Tsigas P., Fouchal H. (eds) Principles of Distributed Systems. OPODIS 2007. Lecture Notes in Computer Science, vol 4878. Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-77096-1_29

[17] C. Evequoz, "Non-Blocking Concurrent FIFO Queues with Single Word Synchronization Primitives," 2008 37th International Conference on Parallel Processing, Portland, OR, 2008, pp. 397-405. https://ieeexplore.ieee.org/abstract/document/4625874

[18] Ladan-Mozes E., Shavit N. (2004) “An Optimistic Approach to Lock-Free FIFO Queues”. In: Guerraoui R. (eds) Distributed Computing. DISC 2004. Lecture Notes in Computer Science, vol 3274. Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-30186-8_9

[19] Maurice Herlihy, Nir Shavit, “The Art of Multiprocessor Programming”, Morgan Kaufmann Publishers Inc, 340 Pine Street, Sixth Floor, San Francisco, CA, United States, ISBN:978-0-12-397337-5. https://dl.acm.org/doi/book/10.5555/2385452

[20] Chien-Hua Shann, Ting-Lu Huang and Cheng Chen, "A practical nonblocking queue algorithm using compare-and-swap," Proceedings Seventh International Conference on Parallel and Distributed Systems (Cat. No.PR00568), Iwate, Japan, 2000, pp. 470-475. https://ieeexplore.ieee.org/abstract/document/857731

[21] Chaoran Yang, John Mellor - Crummey, “A wait-free queue as fast as fetch-and-add”, ACM SIGPLAN Notices, February 2016, Article No.: 16. https://doi.org/10.1145/3016078.2851168

[22] David M. (2004) ”A Single-Enqueuer Wait-Free Queue Implementation”. In: Guerraoui R. (eds) Distributed Computing. DISC 2004. Lecture Notes in Computer Science, vol 3274. Springer, Berlin, Heidelberg.
https://link.springer.com/chapter/10.1007/978-3-540-30186-8_10

[23] Alex Kogan, Erez Petrank, “Wait-free queues with multiple enqueuers and dequeuers”, ACM SIGPLAN Notices, February 2011. https://doi.org/10.1145/2038037.1941585

[24] Hakan Sundell, Philipas Tsigas, “Lock-free deques and doubly linked lists”, in Journal of Parallel and Distributed Computing, Volume 68, Issue 7, July 2008, Pages 1008 - 1020.
https://www.sciencedirect.com/science/article/pii/S0743731508000518?casa_token=LR-x_KZbMM8AAAAA:ZwH4Yfq8aurfv4cBqW1LUX2jQMiFG4vxg1Lakj5_Kq1Le1PWTgvrhPsi0gJHJIvtcAiDfw

[25] Braginsky A., Petrank E. (2011) “Locality-Conscious Lock-Free Linked Lists”. In: Aguilera M.K., Yu H., Vaidya N.H., Srinivasan V., Choudhury R.R. (eds) Distributed Computing and Networking. ICDCN 2011. Lecture Notes in Computer Science, vol 6522. Springer, Berlin, Heidelberg.
https://link.springer.com/chapter/10.1007/978-3-642-17679-1_10

[26] Mikhail Fomitchev, Eric Ruppert, “Lock-free linked lists and skip lists”. In: PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, July 2004, Pages 50–59.. https://doi.org/10.1145/1011767.1011776

[27] John D. Valois, “Lock-free linked lists using compare-and-swap”. In PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, August 1995, Pages 214–222. https://doi.org/10.1145/224964.224988

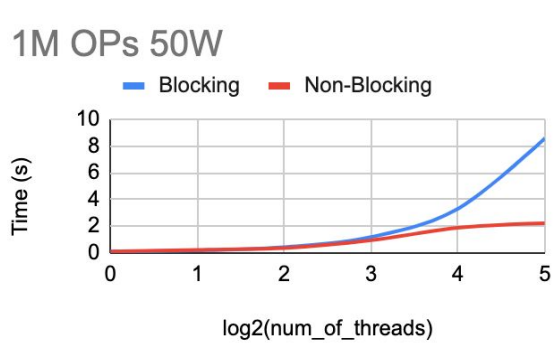
[28] Timnat S., Braginsky A., Kogan A., Petrank E. (2012) “Wait-Free Linked-Lists”. In: Baldoni R., Flocchini P., Binoy R. (eds) Principles of Distributed Systems. OPODIS 2012. Lecture Notes in Computer Science, vol 7702. Springer, Berlin, Heidelberg.
https://link.springer.com/chapter/10.1007/978-3-642-35476-2_23

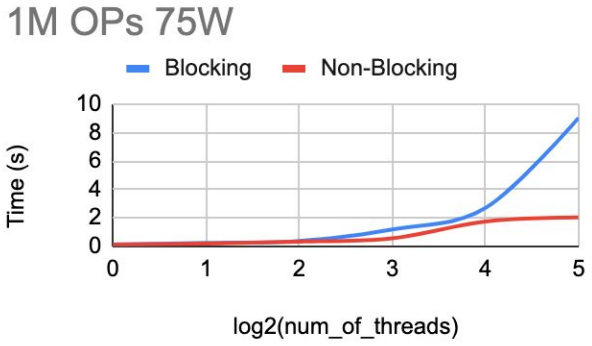
[29] Alex Kogan, Erez Petrank, “A methodology for creating fast wait-free data structures”. In ACM SIGPLAN Notices, February 2012. https://doi.org/10.1145/2370036.2145835

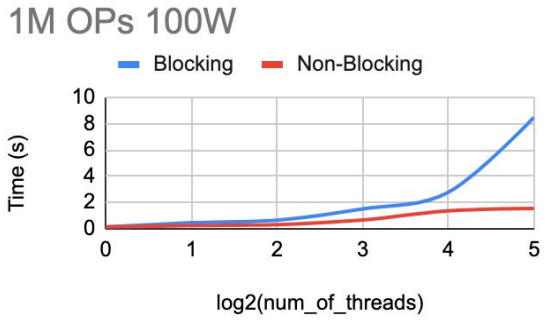
[30] Yehuda Afek, Dalia Dauber, Dan Touitou “Waif-free made fast”. In STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing, May 1995, Pages 538–547. https://doi.org/10.1145/225058.225271

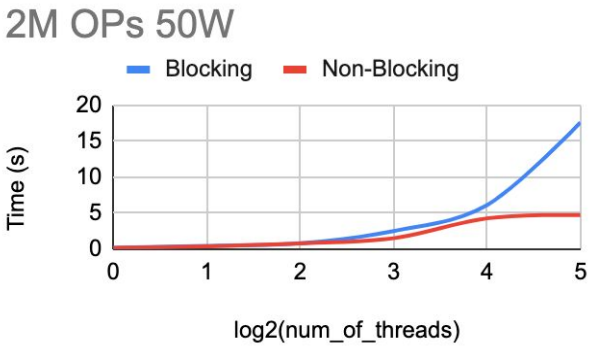
8 Appendix

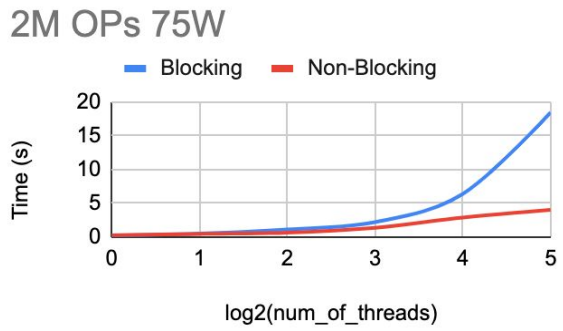
The diagrams below are the comparison between blocking and non-blocking queue in our experiment:

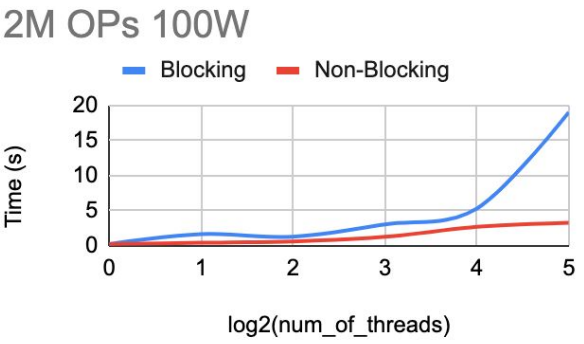


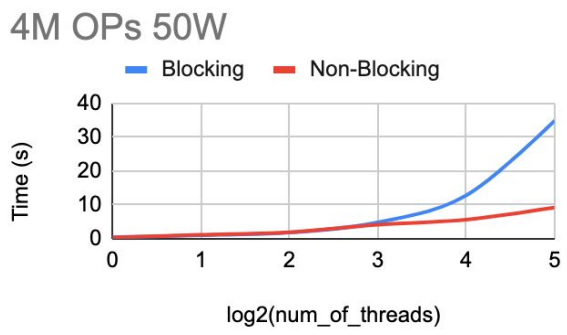


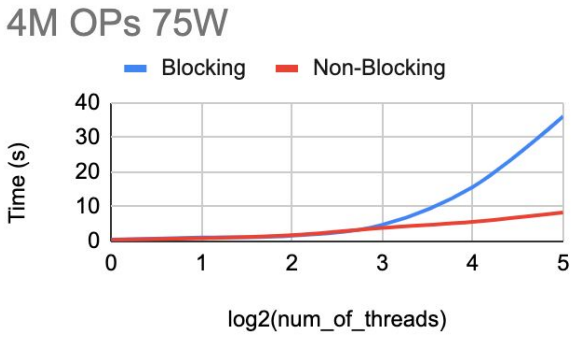


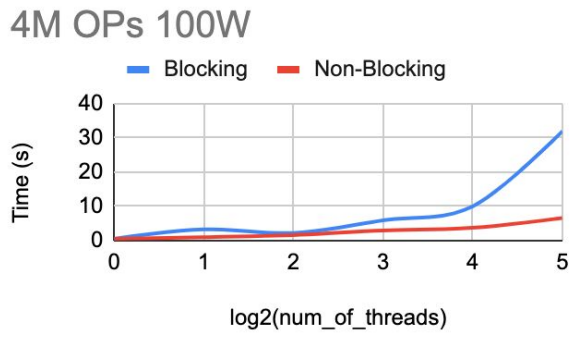


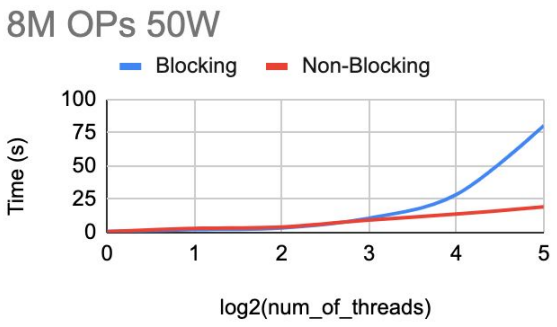


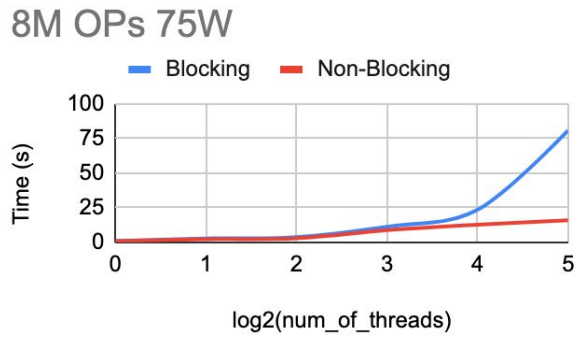


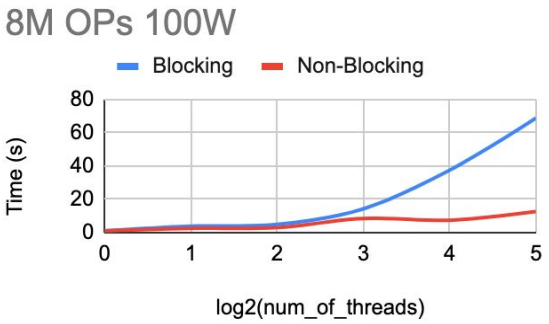


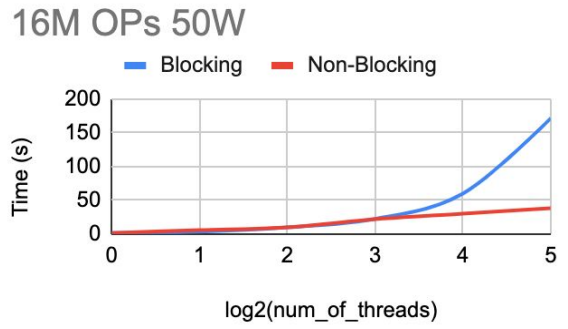


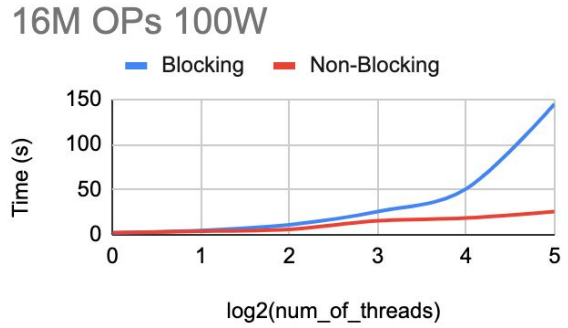






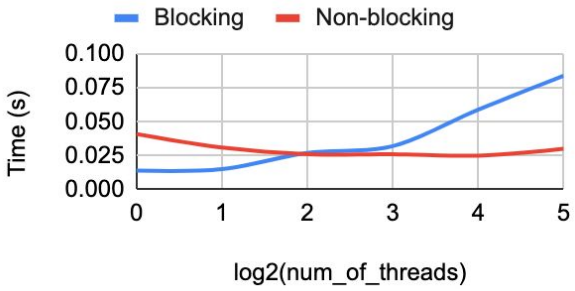




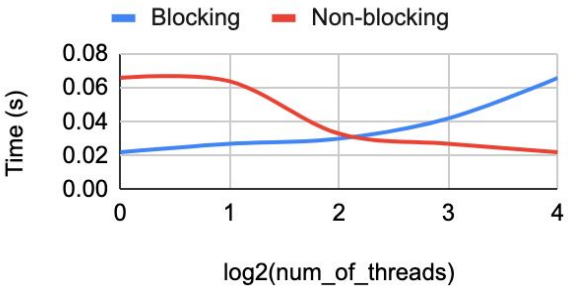


The diagrams below are the comparison between blocking and non-blocking linked-list in our experiment:

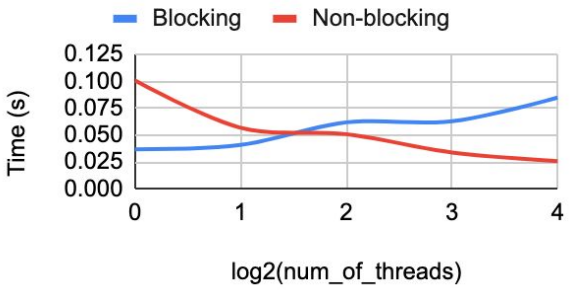
5K OPs 30W



5K OPs 50W

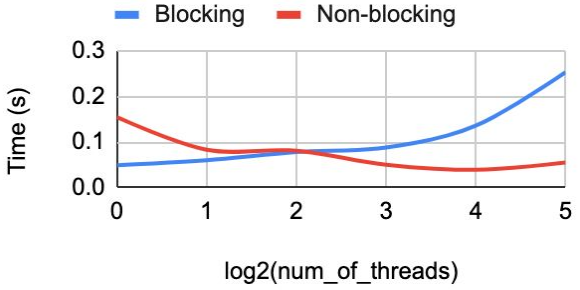


5K OPs 75W

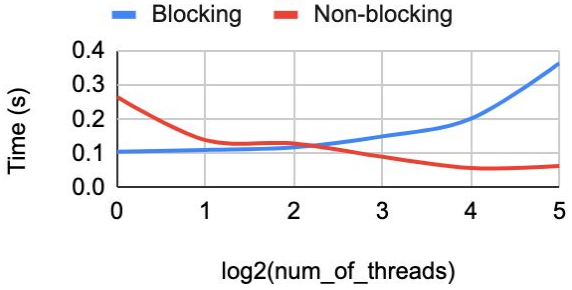


5K OPs 100W

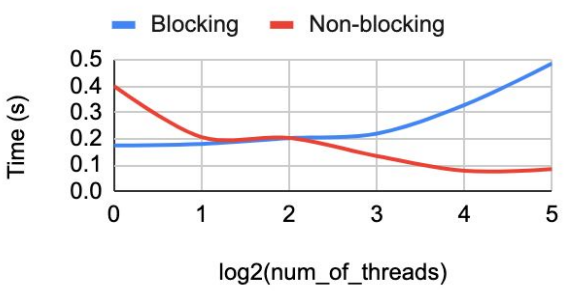
10K OPs 30W



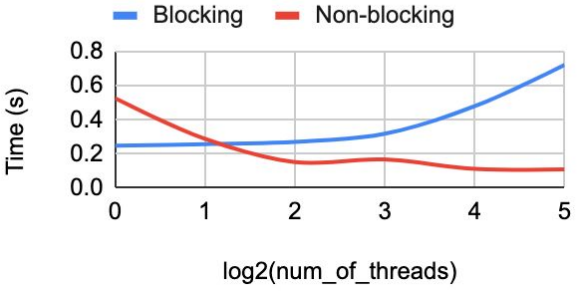
10K OPs 50W



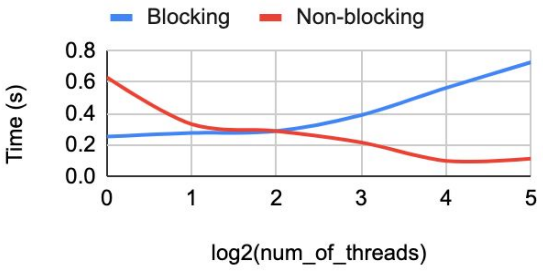
10K OPs 75W



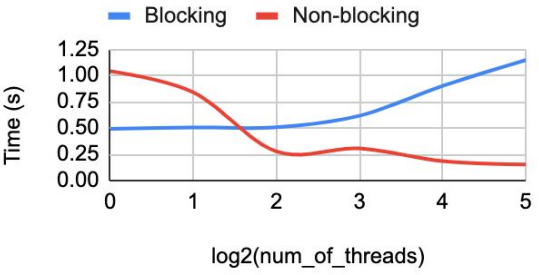
10K OPs 100W



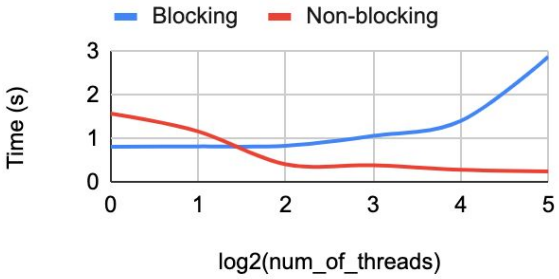
20K OPs 30W



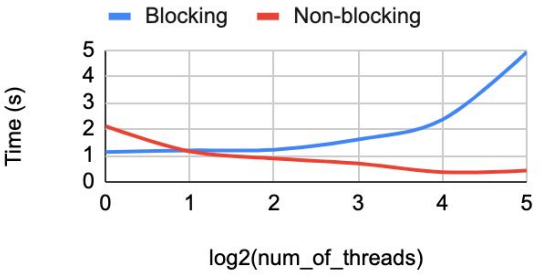
20K OPs 50W



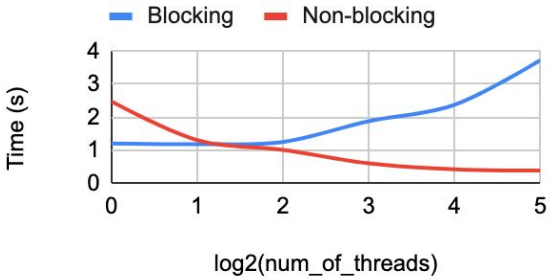
20K OPs 75W



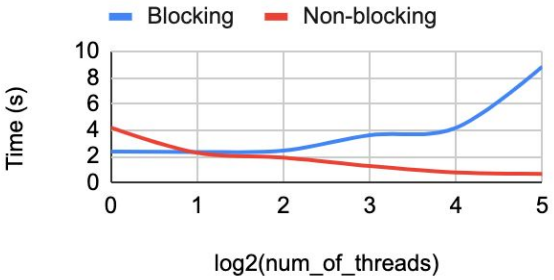
20K OPs 100W



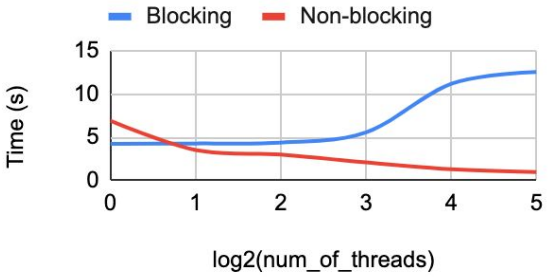
40K OPs 30W



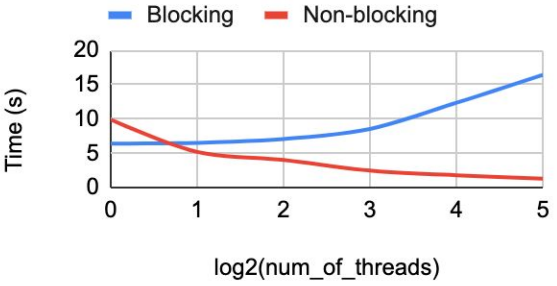
40K OPs 50W



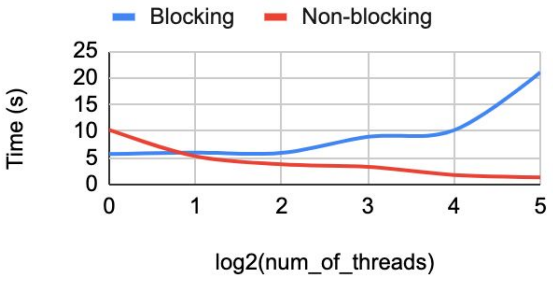
40K OPs 75W



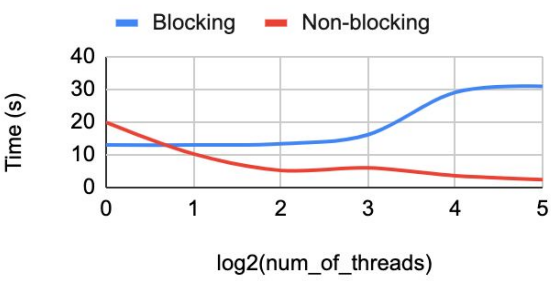
40K OPs 100W

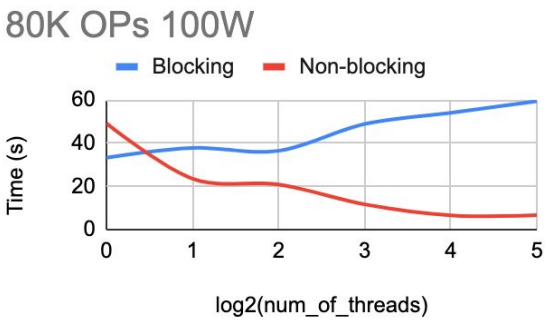


80K OPs 30W



80K OPs 50W





The tables below are the whole data of the experiments, row means data size and column means the number of threads:

Table 1: The experiment data of blocking queue in 50% write and 50% read

	1	2	4	8	16	32
1000000	0.099	0.185	0.431	1.182	3.3	8.573
2000000	0.192	0.44	0.806	2.514	6.13	17.624
4000000	0.387	1.019	1.807	4.825	12.778	34.884
8000000	0.776	2.157	3.397	10.772	28.522	80.343
16000000	1.586	3.412	9.823	22.295	59.811	171.461

Table 2: The experiment data of blocking queue in 75% write and 25% read

	1	2	4	8	16	32
1000000	0.122	0.234	0.369	1.181	2.696	9.024
2000000	0.238	0.504	1.111	2.204	6.388	18.459
4000000	0.468	1.062	1.608	4.766	15.696	36.067
8000000	0.963	2.794	3.697	11.272	23.496	80.911
16000000	1.87	3.966	13.427	23.09	60.624	172.49

Table 3: The experiment data of blocking queue in 100% write and 0% read

	1	2	4	8	16	32
1000000	0.137	0.452	0.652	1.508	2.771	8.48
2000000	0.26	1.662	1.297	3.058	5.319	19.04
4000000	0.52	3.267	2.199	5.89	9.968	31.918
8000000	1.031	3.805	4.857	14.22	37.494	68.835
16000000	2.072	4.705	11.055	25.649	50.878	144.795

Table 4: The experiment data of non-blocking queue in 50% write and 50% read

	1	2	4	8	16	32
1000000	0.097	0.221	0.366	0.941	1.877	2.22
2000000	0.182	0.377	0.792	1.509	4.28	4.739
4000000	0.364	1.133	1.917	4.132	5.637	9.226
8000000	0.727	3.059	4.126	9.403	13.869	19.24
16000000	1.452	5.683	9.86	21.975	30.022	38.362

Table 5: The experiment data of non-blocking queue in 75% write and 25% read

	1	2	4	8	16	32
1000000	0.114	0.188	0.331	0.557	1.745	2.035
2000000	0.228	0.457	0.634	1.372	2.883	4.029
4000000	0.451	0.881	1.754	3.872	5.627	8.346
8000000	0.904	2.243	2.889	8.771	12.745	15.945
16000000	1.788	3.746	7.368	20.961	24.585	34.739

Table 6: The experiment data of non-blocking queue in 100% write and 0% read

	1	2	4	8	16	32
1000000	0.13	0.232	0.311	0.664	1.36	1.537
2000000	0.26	0.431	0.623	1.277	2.699	3.283
4000000	0.513	0.958	1.592	2.943	3.712	6.564
8000000	1.009	2.52	2.934	8.465	7.39	12.588
16000000	1.983	3.829	5.905	15.45	18.439	25.646

Table 7: The experiment data of blocking linked list in 100% write and 0% read

	1	2	4	8	16	32
5000	0.053	0.06	0.062	0.071	0.114	0.202
10000	0.248	0.257	0.27	0.318	0.48	0.723
20000	1.131	1.195	1.222	1.607	2.358	4.914
40000	6.421	6.529	7.084	8.556	12.362	16.443
80000	33.301	37.83	36.499	48.913	54.093	59.417

Table 8: The experiment data of blocking linked list in 75% write and 25% read

	1	2	4	8	16	32
5000	0.037	0.041	0.062	0.063	0.085	0.171

10000	0.175	0.181	0.203	0.22	0.329	0.487
20000	0.814	0.821	0.836	1.061	1.405	2.878
40000	4.235	4.276	4.385	5.562	11.198	12.573
80000	22.226	22.676	26.407	33.944	40.054	43.666

Table 9: The experiment data of blocking linked list in 50% write and 50% read

	1	2	4	8	16	32
5000	0.022	0.027	0.03	0.042	0.066	0.121
10000	0.105	0.11	0.118	0.15	0.202	0.365
20000	0.492	0.506	0.507	0.617	0.901	1.147
40000	2.385	2.355	2.45	3.632	4.193	8.859
80000	13.022	12.968	13.333	16.099	29.002	30.808

Table 10: The experiment data of blocking linked list in 30% write and 70% read

	1	2	4	8	16	32
5000	0.014	0.015	0.027	0.032	0.059	0.084
10000	0.049	0.06	0.078	0.088	0.136	0.254
20000	0.254	0.277	0.289	0.391	0.564	0.727
40000	1.205	1.185	1.253	1.882	2.376	3.728
80000	5.725	6.008	5.946	8.963	10.201	21.046

Table 11: The experiment data of non-blocking linked list in 100% write and 0% read

	1	2	4	8	16	32
5000	0.134	0.088	0.065	0.053	0.029	0.038
10000	0.527	0.288	0.151	0.167	0.111	0.109
20000	2.113	1.158	0.891	0.698	0.375	0.431
40000	9.907	5.204	4.028	2.482	1.814	1.312
80000	49.189	23.589	20.938	11.743	6.608	6.75

Table 12: The experiment data of non-blocking linked list in 75% write and 25% read

	1	2	4	8	16	32
5000	0.101	0.057	0.051	0.034	0.026	0.031
10000	0.401	0.207	0.204	0.135	0.079	0.085
20000	1.575	1.165	0.409	0.39	0.286	0.248
40000	6.911	3.523	2.985	2.084	1.289	0.976

80000	32.324	17.089	8.377	7.536	4.968	3.267

Table 13: The experiment data of non-blocking linked list in 50% write and 50% read

	1	2	4	8	16	32
5000	0.066	0.064	0.033	0.027	0.022	0.033
10000	0.266	0.139	0.129	0.09	0.057	0.063
20000	1.041	0.84	0.277	0.306	0.185	0.153
40000	4.204	2.283	1.922	1.277	0.796	0.684
80000	19.88	10.257	5.218	6.015	3.621	2.41

Table 14: The experiment data of non-blocking linked list in 30% write and 70% read

	1	2	4	8	16	32
5000	0.041	0.031	0.026	0.026	0.025	0.03
10000	0.155	0.083	0.081	0.05	0.039	0.055
20000	0.632	0.334	0.289	0.217	0.1	0.114
40000	2.482	1.309	1.013	0.605	0.423	0.389
80000	10.286	5.326	3.809	3.341	1.828	1.39