Advanced Multiprocessor Programming
Project Group 2

Alexander Redl, 01629061

January 5, 2026

1 Exercise 1

The sequential queue implementation can be found in `src/seq.c`, and all function and structure definitions in the `src/queue.h` header file. The header file was introduced for convenience, such that all following queue implementations operate with the same function signatures.

The type of `value_t` was chosen as `int`, but can easily be adapted `src/queue.h` in the line

```c
typedef int value_t;
```

The implementations of `enq()` and `deq()` follow the lecture and dequeued elements are put in a thread local free list that is checked and, if possible, used in enquing.

The defined types for the sequential implementation are `node`

```c
typedef struct node {
    value_t value;
    struct node *next;
} node;
```

which is a singly linked list node with a value and a pointer to the next node.

And `queue`, defined as

```c
typedef struct queue {
    node *head;
    node *tail;
    node **freelists;
    int max_threads;
} queue;
```

which has a pointer to the sentinel node (`head`), a pointer to the last node in the list (`tail`), a pointer to a list of thread local free lists for recycling dequed nodes (`freelists`), as well as the amount of free lists (`max_threads`), which will be equal to the maximum number of threads.

A queue can be created using

```c
queue* create();
```

which facilitates the usage of different queue structures later on.

Initialization is done using

```c
int init(queue *q);
```

which creates a sentinel node and sets the pointers of the queue `head` and `tail` to this node. It also allocates the free lists and sets `max_threads` = `omp_get_max_threads()`. It is not a concurrent function.

It returns a specific return code that can be checked against special return codes:

- `QUEUE_OK`: the operation was successful
- `QUEUE_NOMEM`: we ran out of memory (for allocations)
- `QUEUE_EMPTY`: returned by `deq()` if the queue is empty

There also exists an helper function

```c
char* q_error(int code);
```

which converts the queue error codes into a user friendly string.

Destroying the queue is done using

```
void destroy(queue *q);
```

which frees all nodes in the queue like in a linked list fashion (so starting from the head and freeing every next link iteratively), and also does the same freeing to each of the free lists. It is not a concurrent function.

The enqueue operation

```
int enq(value_t v, queue *q);
```

allocates a new node or reuses one from the thread local free list, and also updates the tail pointer of the queue. It returns one of the return codes.

The dequeue operation

```
int deq(value_t *v, queue *q);
```

updates the queue head and moves the old sentinel to the local free list for reuse. It sets the reference passed value `v` to the head->next value, and returns one of the return codes.

There also exist enqueue and dequeue operations that additionally take a statistic structure as argument, such that we can benchmark some statistics.

```
int enq_stats(value_t v, queue *q, stats *s);
int deq_stats(value_t *v, queue *q, stats *s);
```

Although not required, I also implemented the operation

```
int len(queue *q);
```

which returns the current length of the queue. This function is never implemented in a thread safe way, as it is only used for testing purposes.

A short test and example usage can be found in `src/test.c` in the `main()` section.

```
queue *q = create();
value_t v;
init(q);
for (int i = 0; i < N; i++) {
  enq((value_t)i, q);
}
printf("%d\n", len(q));
for (int i = 0; i < N; i++) {
  deq(&v, q);
}
printf("%d\n", len(q));
destroy(q);
```

It also includes a `test_seq()` function, checking for sequential correctness and it **is passed** by this implementation. A second short test in the `test_conc()` function checks for concurrent correctness and **is not passed** by this implementation - as expected.

This testing can be compiled and executed using

```
make test_seq
```

Testing the other implementations is done the same way. Additionally one can run a correctness benchmark using

```
make
build/bench_seq -c -n 1
```

which **is passed** for one thread, but **is not passed** for more than one thread - as expected.

2 Exercise 2

The concurrent implementation from the sequential queue using a lock can be found in `src/conc.c`.

The `queue` structure changed to

```c
typedef struct queue {
    node *head;
    node *tail;
    node **freelists;
    int max_threads;
    omp_lock_t lock;
} queue;
```

and now includes the `OpenMP` lock, as we already use `OpenMP` in this project. We could also use a self-written lock for N-threads, but this was already introduced in the exercises, so it is not shown here. I use the functions `omp_init_lock()` to initialize the lock in the `init()` function, `omp_set_lock()` at the start of `enq()` and `deq()` to only allow to enter these functions if the lock is obtained, `omp_unset_lock()` on return of these two functions to ensure the lock is freed again, and `omp_destroy_lock()` in `destroy()` to destroy the lock when the queue is destroyed. The functions `init()`, `destroy()` and `len()` are not protected by the lock and are not concurrent implementations, which is fine according to task 1.

Using my tests (`make test_conc`), the sequential correctness test **is passed**, and the concurrent correctness test **is passed** by this implementation.

One can run a correctness benchmark using

```
make
build/bench_conc -c
```

which **is passed** by this implementation. This correctness benchmark also works for the other implementations in an analog way.

3 Exercise 3

The benchmark code is found in `src/bench.c`. It works with all implementations of the queue and has to be linked accordingly. To facilitate this, one can run the following commands to build all benchmarks and execute them

```
make b_bench
build/bench_seq #or _conc, _conc2, _cas
```

It can be passed multiple parameters

- `-n`: number of threads (default is `omp_get_max_threads()`)
- `-t`: time in seconds for each experiment (default is 1)
- `-r`: number of repetitions (default is 1)
- `-c`: correctness check, which will ignore all flags except -n and -t
- `-h`: help and usage, which basically prints this here
- `-e`: enqueue batch size as fixed size (`-e 10`), or as randomly chosen from a range (`-e 5,10`)
- `-d`: dequeue batch size as fixed size (`-d 10`), or as randomly chosen from a range (`-d 5,10`)
- `-E`: enqueue batch size per thread (e.g. -E 10,0), which length must match to the value of -n
- `-D`: dequeue batch size per thread (e.g. -D 0,10), which length must match to the value of -n

It will store different statistics (duration, enqueue successes, enqueue errors, dequeue successes, dequeue errors, free list inserts, maximum free list size, CAS successes, CAS errors) per thread and print them along with the overall (averaged or summed, depending on the variable) statistics.

Especially the correctness test (`-c`) is useful when checking the queue implementations for correctness. As given in the assignment: For $\frac{(\texttt{-t})}{2}$ seconds every thread enqueues thread-unique values into the queue. It is logged how many numbers each thread enqueues. On dequeuing, every thread can therefore identify which thread had enqued the dequed number and a counter is increased. If the enque number and deque counter number per thread match, the implementation is (practically) correct.

4 Exercise 4

The concurrent implementation from the sequential queue using two locks (one for the `enq()` and one for the `deq()` operation) can be found in `src/conc2.c`.

The `queue` structure changed to

```
typedef struct queue {
    node *head;
    node *tail;
    node **freelists;
    int max_threads;
    omp_lock_t lock_enq;
    omp_lock_t lock_deq;
} queue;
```

and now includes two of the `OpenMP` locks. According to their name, one is used at the entry and return of the `enq()` operation, and one on the `deq()` operation.

Using my tests (`make test_conc2`), the sequential correctness test **is passed**, and the concurrent correctness test **is passed** by this implementation. The correctness benchmark (`build/bench_conc2 -c`) **is passed** as well.

The memory management in terms of the free list continues to work in the same procedure: as each thread has its own local free list, an access to this free list can only occur by the corresponding thread. As per thread the function calls (here `enq()` and `deq()`) are sequential, there are no overlapping memory accesses and no unexpected behavior. Due to the locking of `deq()` for all threads, it is not possible that a thread wants to deque a node that has already be put in a freelist or even again been enqueud in the queue by another thread.

The ABA problem is prevented by mutual exclusion, as if a thread falls asleep on the dequeuing, its gathered lock will prevent any other thread from performing the dequeue operation, so the head node will stay the same. The same holds true for any enqueuing and the tail node. Here, also the lock prevents any other thread from enqueuing, meaning the modification of nodes after the tail.

The implementation is linearizable, because each operation can be assigned a specific timepoint when it affects the state of the queue. The `enq()` operation is linearized when

```
q->tail = n;
```

is updated, and the `deq()` operation is linearized when

```
q->head = new;
```

is updated.

5 Exercise 5

The concurrent implementation using compare-and-swap (CAS) operations can be found in `src/cas.c`. It took heavy inspiration from the Michael-Scott implementation mentioned in the lecture notes [Maged M. Michael, Michael L. Scott: Simple, Fast, and Practical Non Blocking and Blocking Concurrent Queue Algorithms. PODC 1996: 267 275].

Instead of pointers to next elements in the linked list of `nodes`, it uses stamped pointers (`snode_ptr`), defined as

```
typedef uint64_t snode_ptr;
```

As in the `x86` architecture we have 64-bit wide addresses, where currently only the lower 48 bits are used, the upper 16 bits are used here as a stamp (`stamp_t`).

```
typedef uint16_t stamp_t;
```

For convenience I defined functions to convert between `node` pointers, `snode_ptr` objects and `stamp_t` stamps.

```
snode_ptr stamp(node *n, stamp_t stamp);
stamp_t get_stamp(snode_ptr sn);
node *get_node(snode_ptr sn);
```

The defined `node` structure changed to

```c
typedef struct node {
    value_t value;
    _Atomic(snode_ptr) snext;
} node;
```

where the pointer to the next `node` is now stored in the atomic `snode_ptr` object `snext`. The atomicity was chosen that CAS operations can work on this object correctly.

The `queue` structure is now defined as

```c
typedef struct queue {
    _Atomic(snode_ptr) head;
    _Atomic(snode_ptr) tail;
    _Atomic(snode_ptr) *freelists;
    int max_threads;
} queue;
```

where all `node` pointers were converted to atomic `snode_ptr` objects, which is (again) required for correct CAS operations.

The `init()`, `destroy()` and `len()` methods still work in the same way as in the other implementations, while `enq()` and `deq()` now follow the lecture notes and the Michael-Scott implementation closely. They still put dequed `nodes` in a thread local freelist and reuse them. No locks are used in this implementation, and the wait-freeness and concurrent correctness is achieved by using CAS operations

```c
#define CAS atomic_compare_exchange_weak
```

Because it uses atomic constructs, such as `_Atomic()`, `atomic_compare_exchange_weak()`, `atomic_load()`, and `atomic_store()`, it requires the C11 standard to be successfully compiled.

Using my tests (`make test_cas`), the sequential correctness test **is passed**, and the concurrent correctness test **is passed** by this implementation. The correctness benchmark (`build/bench_cas -c`) **is passed** as well.

The ABA problem is solved by introducing stamped pointers (`snode_ptr`). The stamp acts as 16 bit counter that is incremented every time the `snode_ptr` object did a successful CAS update. Thus, a CAS operation can not succeed on a `snode_ptr` object (= `node` pointer) that has been removed (and possibly later reinserted), even if the raw address is reused.

Nota bene: Theoretically, the ABA problem still exists, as the 16 bit stamp could overflow and take the exact value it had before it was updated 2^{16} times. So if a thread sleeps for the correct amount of time it could still reference a node with a `snode_ptr` that was increased 2^{16} times and now again matches to the one the thread woke up to. But this is highly unlikely practically speaking ($2^{16} = 65536$) and did not occur to me once during my testing.

6 Exercise 6

The benchmark can be done using

```
make bench
```

which utilizes the `run_nebula_seq.sh` and `run_nebula_conc.sh` scripts to benchmark the implementations on the `nebula` server for the given configurations: 10 repetitions, batch sizes of 1 and 1000, experiment times of 1s and 5s, number of threads in [1,2,8,10,20,32,45,64] and different enque/deque patterns.

- Pattern a: all threads enqueing and dequeuing with the same batch sizes
- Pattern b: one thread (first one) enqueing, all other threads dequeuing
- Pattern c: all threads with id smaller than $\frac{p}{2}$ enqueing only, the other threads dequeuing only
- Pattern d: even numbered threads enqueing, odd numbered threads dequeuing

The benchmark runs for ~4 hours and its output is saved to `data/*.log`. All log files are already provided in the project, but will be deleted if another benchmark is performed.

To visualize the output, a plotting routine in `plot.py` (utilizing `matplotlib`) can be called with

```
make plot
```

which stores its output plots in `plots/*.pdf`. These plots are shown and discussed in the following paragraphs.

The throughput (defined as successful operations (`enq()` and `deq()`) per second) is shown in Fig. 1, Fig. 2, Fig. 3, and Fig. 4. One immediately observes, that the sequential implementation works the best in all scenarios and the concurrent implementation gets worse the more threads there are. This makes sense, as we have high contention on shared atomics (or locked sections) which will reduce throughput. Additionally, we have overhead in the concurrent implementations compared to the sequential one. Also we have a lot of CAS retries in an infinite loop upon success.

The implementation using 2 locks (`conc2`) turns out to have the highest throughput in almost all scenarios, while the one with 1 lock (`conc`) is most of the time, especially for high thread numbers, the worst. As we can run `enq()` and `deq()` concurrently with `conc2` compared to `conc`, this makes sense. The CAS implementation (`cas`) is mostly in between the other two implementations.

For pattern *a* the concurrent throughput seems to be the best in comparison to the other patterns. For pattern *c* the concurrent throughput seems to be the worst. That pattern *a* performs the best is explained as each thread enques and deques, therefore the total number of enques and deques is the same. This is not guaranteed if e.g. one thread only enques and one thread only deques, as they might perform a different amount of operations in a given timeframe. This is exactly observed in patterns *c* and *d*. I would have assumed pattern *b* to perform worst, as only one thread enques, however, it seems that a queue is a bad datastructure to make concurrent, as one thread enqueuing already stretches the queues workload and multiple threads enqueuing does not improve throughput.

Even for 1 thread all concurrent implementations have a throughput that is at least a factor of 3 reduced from the sequential implementation, due to introduced overhead.

The throughput is mostly unaffected by the experiment time (e.g. compare Fig. 1 and Fig. 3). A higher batch size from 1 to 1000 increases the throughput by a factor of roughly 7.

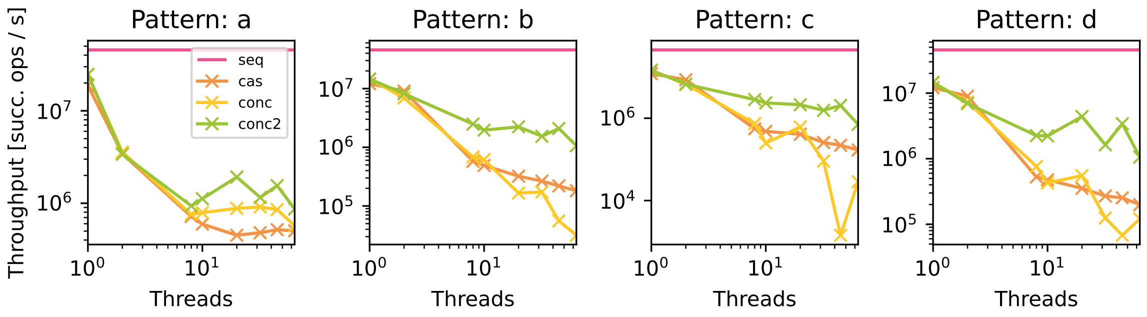


Figure 1: Throughput of successful operations for t=1s and batch size=1.

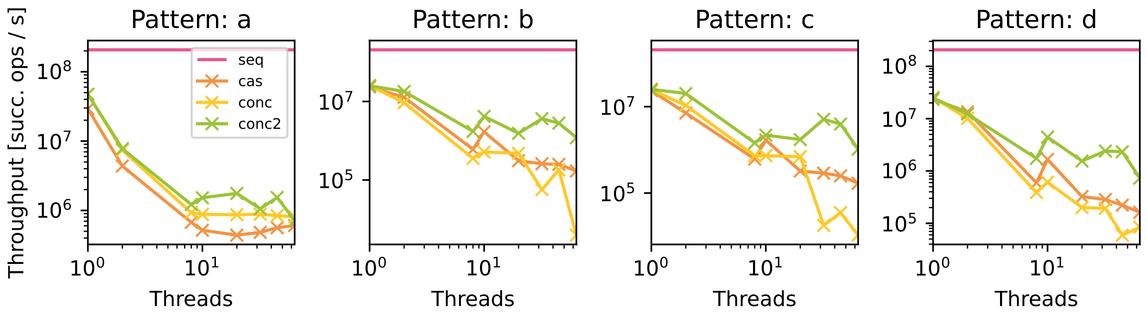


Figure 2: Throughput of successful operations for t=1s and batch size=1000.

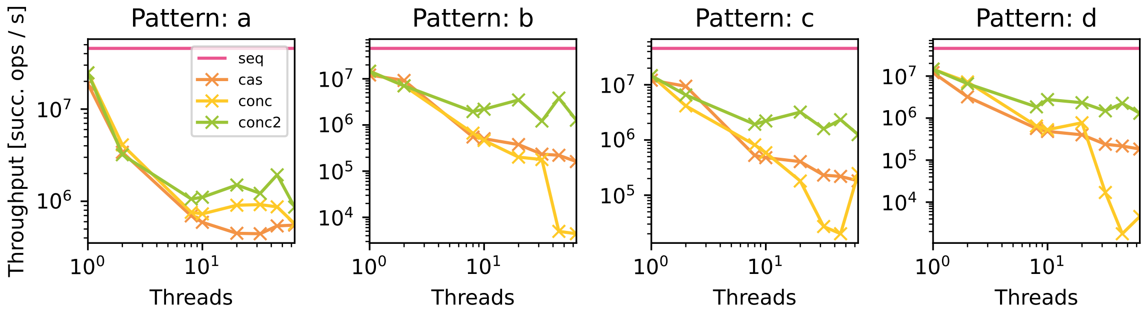


Figure 3: Throughput of successful operations for t=5s and batch size=1.

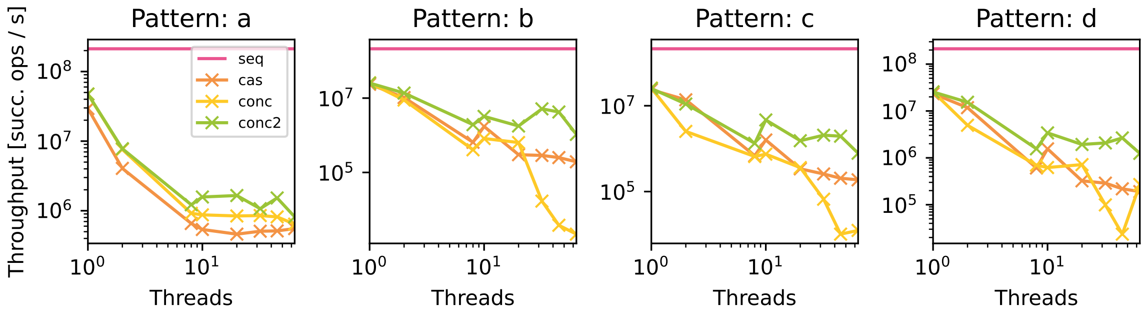


Figure 4: Throughput of successful operations for t=5s and batch size=1000.

The according speedup (or actually *speeddown*) is shown in Fig. 5, Fig. 6, Fig. 7, and Fig. 8. The information from these graphs is identical to the already done analysis on the throughput graphs and therefore discussed above. The higher the thread number, the slower the queue gets.

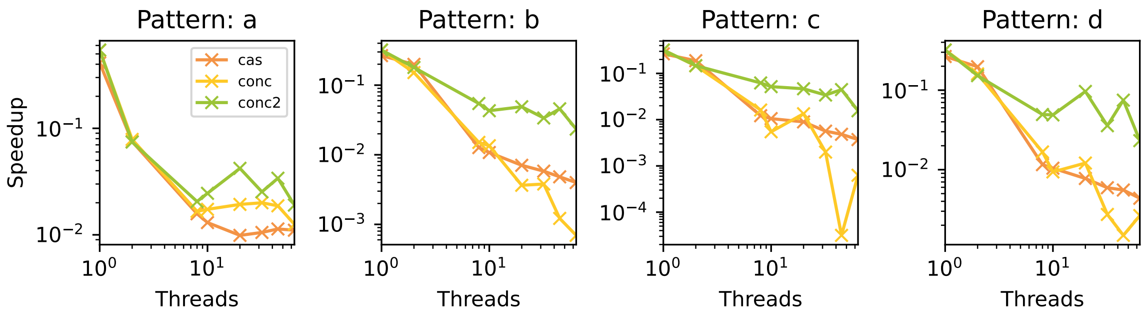


Figure 5: Speedup of successful operations for t=1s and batch size=1.

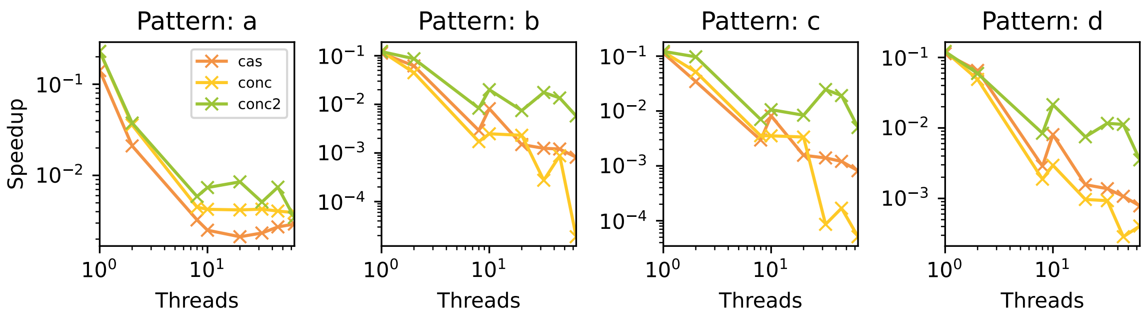


Figure 6: Speedup of successful operations for t=1s and batch size=1000.

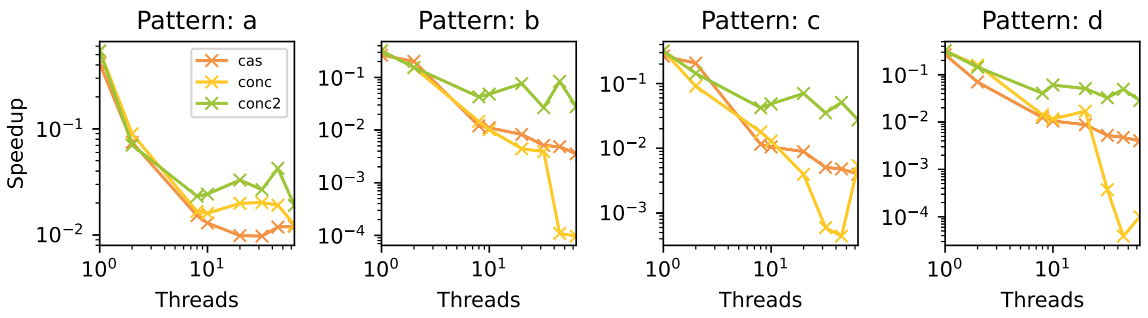


Figure 7: Speedup of successful operations for t=5s and batch size=1.

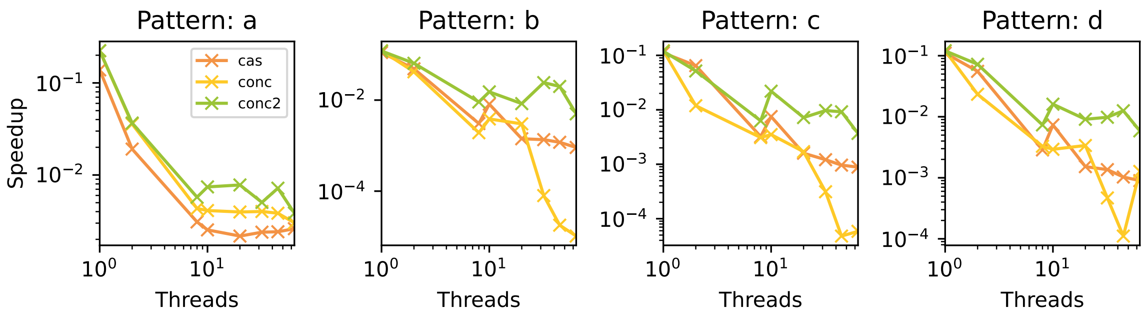


Figure 8: Speedup of successful operations for t=5s and batch size=1000.

Additionally, I looked into the total number of freelist insert operations (summed over all threads), which is shown in Fig. 9, Fig. 10, Fig. 11, and Fig. 12. The number of freelist inserts generally decreases with increasing thread count, as we do not dequeue so many elements in the same timeframe. The `conc2` implementation almost always has the highest number of inserts, which makes sense, as it also has the highest throughput. Of course for t=5s we have ∼5 times more freelist inserts than for t=1s. It is, however, relatively unaffected by the batch size, where I would have expected also a factor of 7 as shown in the throughput graphs and discussion.

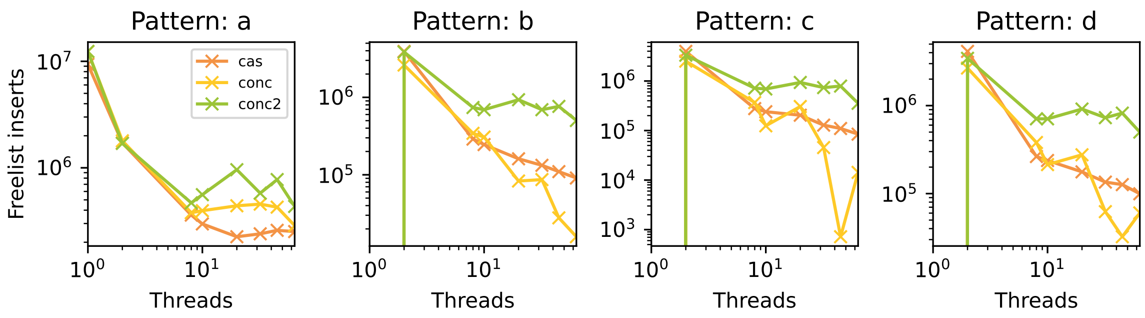


Figure 9: Freelist insert operations for t=1s and batch size=1.

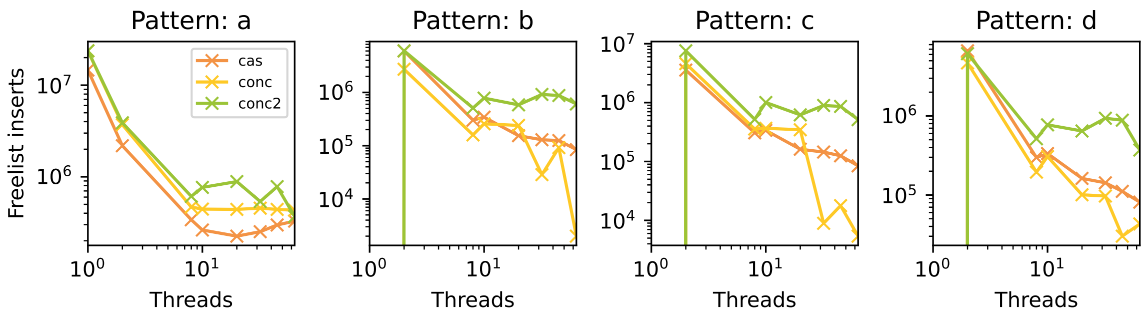


Figure 10: Freelist insert operations for t=1s and batch size=1000.

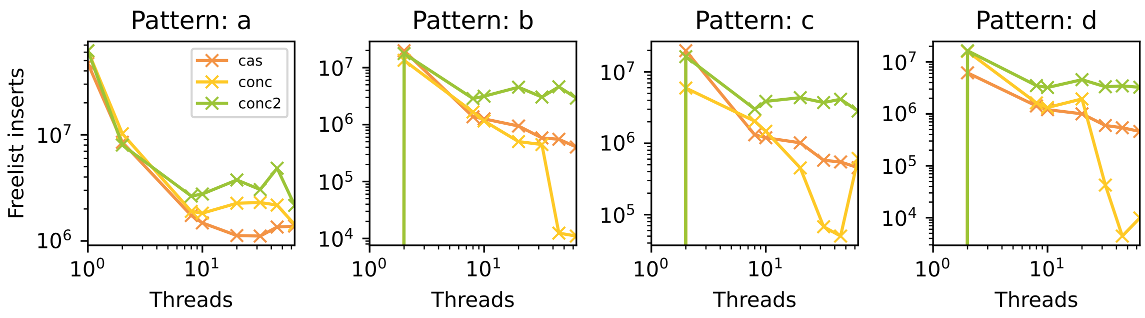


Figure 11: Freelist insert operations for t=5s and batch size=1.

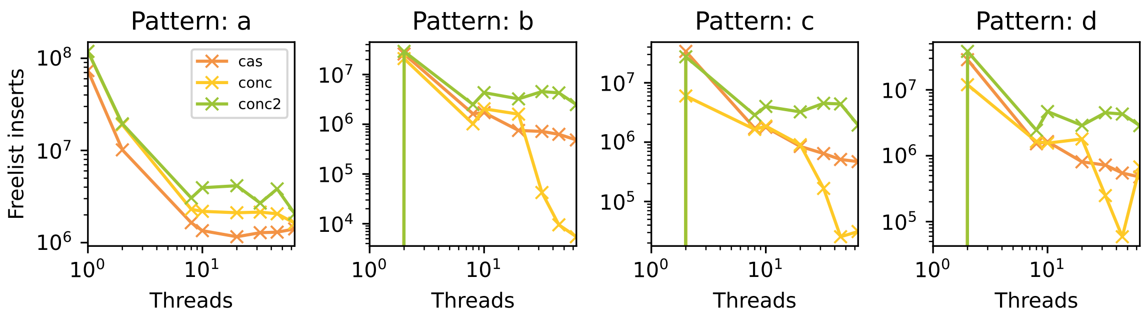


Figure 12: Freelist insert operations for t=5s and batch size=1000.

The maximum length of any thread local freelist is shown in Fig. 13, Fig. 14, Fig. 15, and Fig. 16. As expected, it is the value of batch size for pattern *a*, as we immediately dequeue after enqueuing batch size-amount of elements. For the other patterns, the more threads we have, the fewer elements are also at maximum in a freelist, similar to the inserts into the freelist.

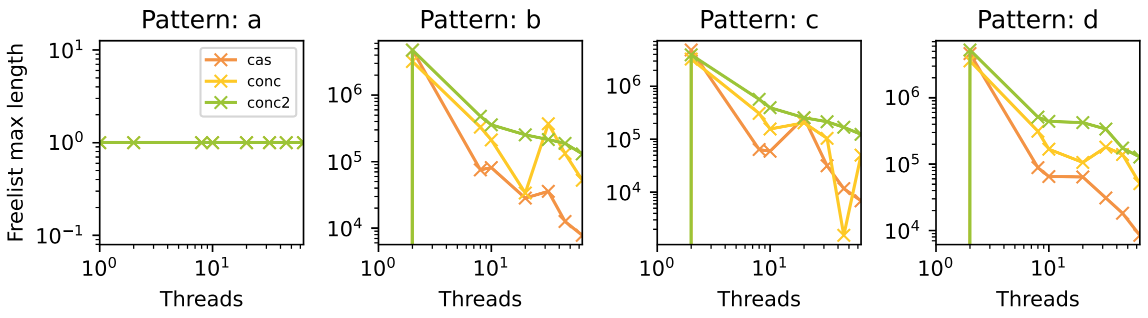


Figure 13: Maximum freelist length for t=1s and batch size=1.

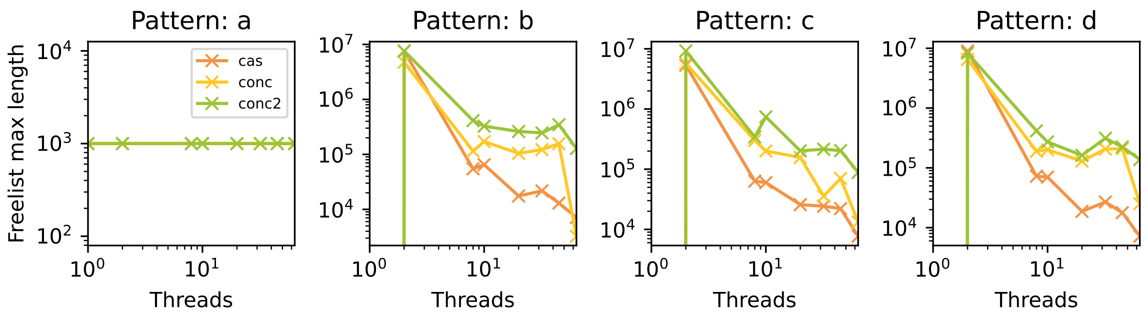


Figure 14: Maximum freelist length for t=1s and batch size=1000.

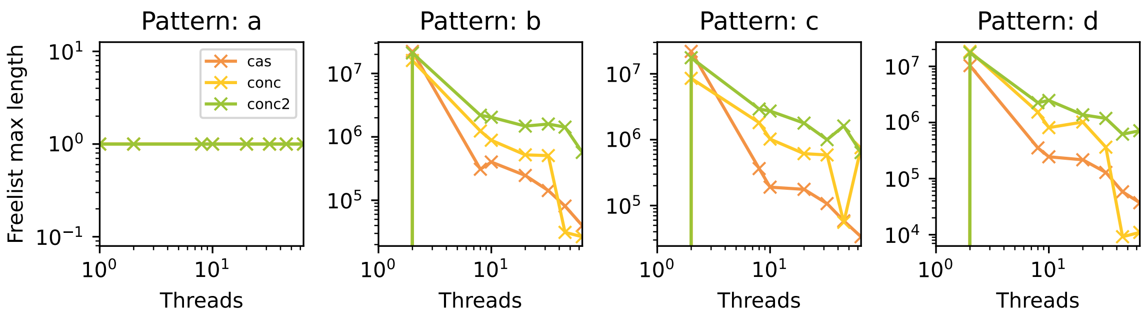


Figure 15: Maximum freelist length for t=5s and batch size=1.

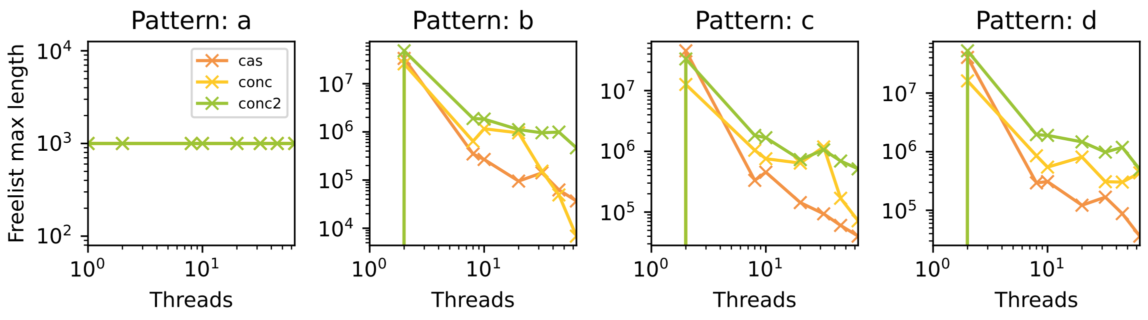


Figure 16: Maximum freelist length for t=5s and batch size=1000.

In Fig. 17, Fig. 18, Fig. 19, and Fig. 20 the failed `deq()` operations are shown. We can see that the CAS implementation has a lot of failed `deq()` operations, especially for the patterns where we do not immediately deque after enqueuing. For pattern *a* I would have expected the dequeue fails to be always 0, which is the case for the lock based implementations and the batch size = 1000 CAS benchmark.

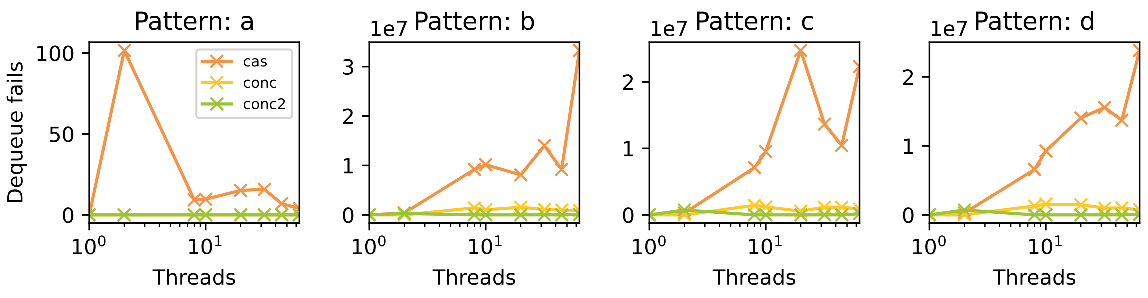


Figure 17: Failed `deq()` operations for t=1s and batch size=1.

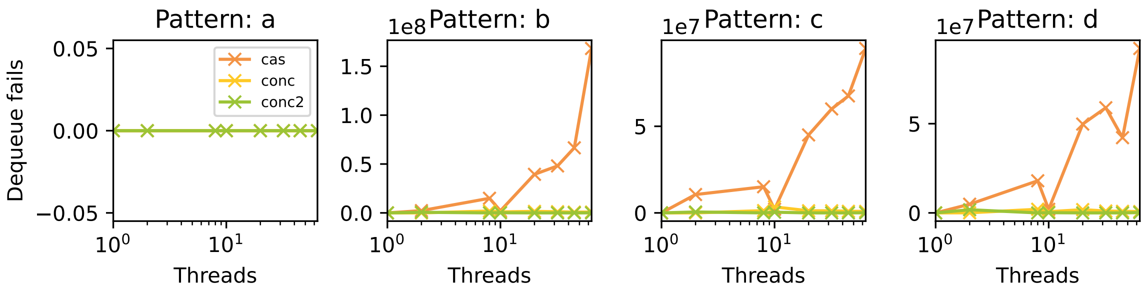


Figure 18: Failed `deq()` operations for t=1s and batch size=1000.

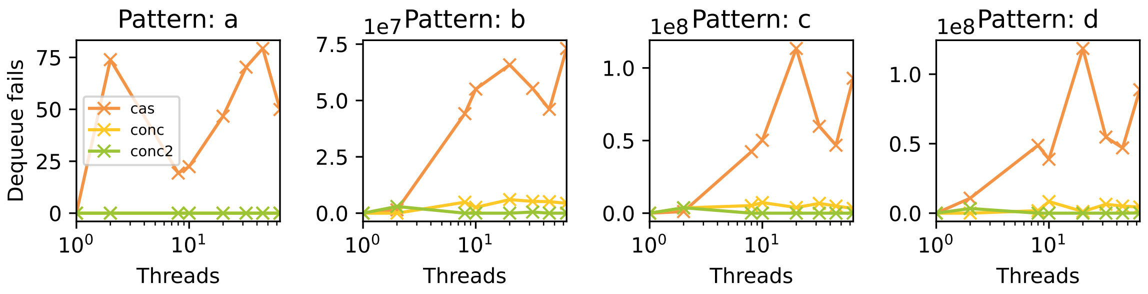


Figure 19: Failed `deq()` operations for t=5s and batch size=1.

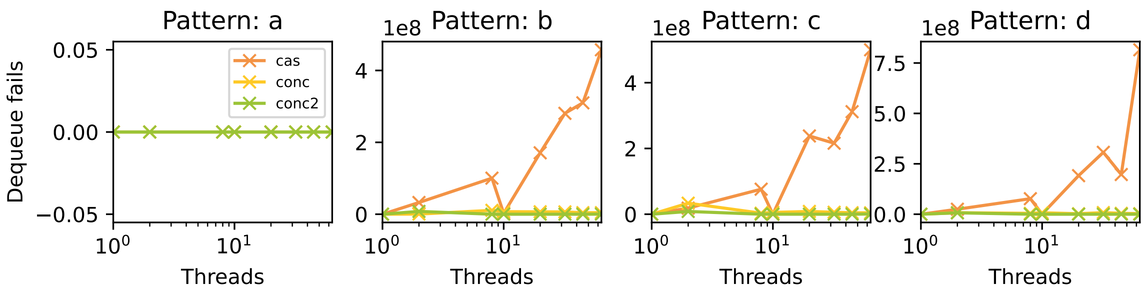
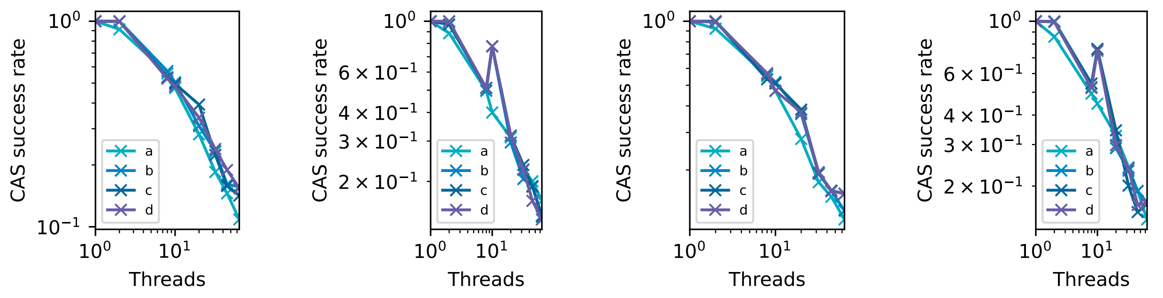


Figure 20: Failed `deq()` operations for t=5s and batch size=1000.

The ratio of successful to total CAS operations is shown in Fig. 21. The more threads, the lower the chance of a successful CAS. All patterns follow the same trend, almost independent of time or batch size. Except for batch size = 1000 and 10 threads, where for patterns *b*, *c*, and *d* we unexpectedly have more successful CAS operations.



t=1, batch size=1 t=1, batch size=1000 t=5, batch size=1 t=5, batch size=1000

Figure 21: Ratio of successful CAS operations.

7 Exercise 7

The definition of linearizability is that method calls should appear to happen in a one-at-a-time ("atomic"), sequential order. Each method call should appear to take effect in real-time-order, instantaneously at some point (the linearization point) between invocation and response event.

For the `enq()` operation, it occurs on a thread local queue. For lock-free queues, each enqueue has a linearization point when the value is successfully linked into the queue (e.g. successful CAS of setting tail->next field). So we have linearizability in the `enq()` operation at the point where it is linearized in the thread local queue.

For a successful `deq()` operation, it occurs on a thread local queue. Again, for lock-free queues, each dequeue has a linearization point (e.g. successful CAS of updating the head). So we have linearizability in the `deq()` operation at the point where it is linearized in the thread local queue.

If the `deq()` operation fails, we cannot assign a linearization point. Assuming thread A tries to dequeue and finds all thread local queues empty, thus the `deq()` operation fails. For a linearization point one needs a specific timepoint when it is true that the bag is empty. Other threads, however, could concurrently enqueue elements after thread A has checked some queues, so it can not be the last local queue check. It also cannot be the first checked local queue, as other threads might enqueue in other local queues, making the bag not empty. Even if thread A has scanned all queues and sees them as empty, another thread could have enqueued before thread A finishes, so at no single timepoint we can guarantee that the bag was empty. Thus, it is not linearizable.

Sequential consistency requires that method calls should appear to happen in a one-at-a-time ("atomic"), sequential order. So for each thread, method calls should appear to take effect in program order. Therefore, it does not require linearization points.

For the `enq()` operation and successful `deq()` operation we have sequential consistency with the same arguments as the linearization. Also, their sequential consistency follows from the underlying correctness of the queues.

Failing the `deq()` operation is still consistent with some sequential ordering. One can e.g. shift the failing dequeue of thread A happened before another threads enqueue, or the other way around. For sequential consistency, the operation can be placed anywhere in the global order, as long as per-thread order is respected. Thus, this can always be found and makes the concurrent bag sequential consistent.

8 File structure

Although I kept the project skeleton structure, I changed a few implementations. One can still use the required commands

```
make zip
bash run_nebula.sh project.zip small-bench
make small-plot
```

However, the `run_nebula.sh` script was rewritten to only execute

```
make small-bench
```

The `small-bench` benchmark tests all 4 implementations, but only with 1 repetition, a time of 1s and batch size of 1000. Additionally, the concurrent implementations only execute on threads of [1,32,64] and pattern *a*. It executed for me in 58-62 seconds.

The file and folder structure is as follows:

- In the `src` folder, all `*.c` source code files and `*.h` header files can be found.
- The `build` folder will contain the built ELF executable files. It is empty upon submission.
- The `data` folder contains the `*.log` benchmark output files for all required configurations.
- The `plots` folder contains all plots created from the benchmark data.
- The `report` folder contains this `report.pdf` report, as well as the report in `report.tex` latex format.
- The `Makefile` contains all important commands for building, testing and benchmarking.
- The `plot.py` python script is used for generating the plots from the benchmark data.
- The `run_nebula.sh` bash script just executes `make small-bench`.
- The `run_nebula_conc.sh` bash script benchmarks a concurrent implementation on the nebula server.
- The `run_nebula_seq.sh` bash script benchmarks a sequential implementation on the nebula server.
- The `README.md` gives a short overview over the project.