Prof. Dr. Jesper Larsson Träff
TU Wien
Faculty of Informatics
Research Group for Parallel Computing

Advanced Multiprocessor Programming (AMP)
WS 2025/26
Programming Project

Issue date: 2025-10-27 (October 27th)
Due date: 2026-01-12 (January 12th)

The AMP programming project deals with implementation in C (or mild C++) with OpenMP (or threads) and benchmarking of *concurrent queues* as described in the AMP lecture (slides on TUWEL, see also [1, 2, Chapter 10]). You are asked to give three different, concurrent implementations of an unbounded FIFO queue data structure implemented as a linked list with a circulating sentinel (head). Correct memory management by thread local free lists is of importance (no ABA problem). The implementations have to be carefully benchmarked against each other.

The hand-in will consist of a short report on the implementations with a performance analysis including plots with your benchmark results, as well as your code in a readable form that will allow to reproduce your findings. The final benchmarks have to be done on the "nebula" system with 64 threads running on all cores. It is recommended to start small using laptop or other small system for development. And it is recommended to **start early**!

The project can be done in **groups of three** (3). Groups (regardless of size) **must** register in TUWEL. The exercise points indicate the amount of work estimated to go into the different parts of the project. In addition to your benchmarks you provide a *small benchmark* that runs in at most *1 minute* and is representative of your results. We use the make-target *small-bench* to automatically verify your implementation.

Problem specification:

Following the algorithms from the lecture, the first task is to implement, in C (or mild C++) a sequential version of the unbounded queue. The implementation must support an

```
void enq(value_t v, queue Q);
```

enqueue operation (here of values of type `double`) and a

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

dequeue operation, and must be based on a linked list with `head` and `tail` pointers, the `head` always pointing to a (changing) sentinel, whose `value_t` value is irrelevant (not used). The value type `value_t` is defined by you, for the correctness checking it may be convenient to let it be an `int`. Both enqueue and dequeue operations shall be total, that is, must always return a value (which could indicate failure). The dequeue operation shall return true or false

(1 or 0) to indicate whether the queue is non-empty or empty, and in case it was non-empty, update the word pointed to by the `value` pointer. The `queue` type (defined by you) represents the queue. Memory management is important to avoid leaking. The queue operations shall internally manage a free list(s), into which dequeued list elements are put and reused. Some algorithms and implementations might have an ABA problem, which you are expected to solve.

Using the sequential implementation as the basis, the first task is to make it concurrent using a single, global lock protecting both the enqueue and the dequeue operations.

The next task is then to implement a benchmark with basic functional correctness checking (to be disabled for the benchmark and production runs). This benchmark shall be used to evaluate all concurrent queue implementations and compare them against each other. The final benchmarking **must** be done on the "nebula" system.

The two next tasks is to develop possibly better (performing) concurrent queues. The first implementation shall use different locks on the `head` and `tail` pointers. The next implementation shall be lock-free and use atomic operations for manipulating the queue list. Both implementations shall be benchmarked against each other and against the simple, global lock concurrent queue.

A final, theoretical task deals with a possible application of the lock-free, concurrent queue to implement a relaxed, unordered pool data structure called a *bag*.

It is recommended to use OpenMP to manage threads (the `#pragma omp parallel` construct, see [3]) and possibly locks and critical sections. For atomics, use the standard C/C++ atomics as discussed in the lecture.

A project skeleton showing how to use OpenMP and how to benchmark and organize the gathered data will be made available on TUWEL and is briefly introduced later.

Exercise 1 (5 points)

Implement a sequential queue with the specified enqueue and dequeue operations based on a singly linked list with `head` and `tail` pointers, using a value type `value_t` and a queue type `queue` defined by you. The implementations can (shall) follow the implementation discussed in the lecture, see also [1, 2, Chapter 10]). Dequeued queue elements shall be put in a free list (associated with the `queue` structure), from which to be inserted elements are taken; when the free list is empty, a new element is globally allocated. Define also an initialization operation and a clean-up operation for your queue (these do not have to be concurrent). Explain briefly the defined types, and very briefly how the implementation is supposed to work. Test (for yourself) as you go along (this will save effort and nerves for the next tasks).

Exercise 2 (5 points)

Make the sequential queue concurrent by using a global lock for all operations (also to be put in the `queue` structure). Mention the lock functionality you used.

Exercise 3 (15 points)

Develop a benchmark for assessing the performance of your concurrent queues, together with possibilities to do (simple) correctness checking (sanity, consistence) of your running queue.

The benchmark runs an experiment a given number of times. An experiment consists of per thread (concurrent) operations on the queue data structure following a given recipe, and carried out within a given time frame (e.g., 1 second, 10 seconds). Thus, a thread ends its experiment when the time limit has been reached.

The benchmark will attempt to use the queue in a balanced way, with each thread enqueueing some elements that are later dequeued. The queue starts empty, and ideally, will end empty as well. More precisely, each thread enqueues a batch of elements, and then dequeues a batch of elements, after which it enqueues the next batch of elements, and so on, until time is up, or a maximum number of enqueue operations has been reached. It must be possible to give enqueue and dequeue batch sizes per thread; batches may not all have the same size, that is, it could/should be possible to specify that batch sizes are chosen randomly, within some range. The setup shall thus make it possible to benchmark use cases, where some threads (say, half of them) enqueue, while the other threads are dequeuing only. A good experiment will be such that over all threads the number of elements in the batches enqueued will match the number of elements in the batches dequeued.

For the performance evaluation, each thread shall record the number of enqueue and dequeue operations it performed, such that a global throughput can be computed. The benchmark shall also record the number of failed (empty) dequeue operations per thread. The benchmark shall, for the different implementations, be instrumented with relevant performance counters, like for instance the number of (failed) compare-and-swap (CAS) operations, the number of elements inserted into the free lists, the maximum size of the free list, and so on. It is important to avoid false-sharing issues with the performance counters (count locally, report globally).

Your benchmark program/procedure must make it possible to define:

- Number of threads.
- Number of repetitions of the experiment.
- Time interval for throughput measurement.
- Enqueue batch size (per thread or according to some pattern)
- Dequeue batch size (per thread or according to some pattern)

Per experiment, the output must be the actual time (average over all threads), total number of operations, the total number of enqueue and successful and failed dequeue operations per thread.

For correctness/consistency, it must be possible to check that all enqueued elements can actually be dequeued, once(!), and that no spurious values appear in the queue: Take a large-enough contiguous interval of (integer) values and allocate for each thread a disjoint subinterval of values to be enqueued. Each thread counts how many values out of its own interval it has enqueued and how many values out of any interval it has dequeued. At the end of

the run, the sum over all dequeued values has to match the number of enqueued elements. Describe briefly your benchmark, what it can do, how to use it, and how it was implemented.

Exercise 4 (10 points)

Implement a concurrent queue with separate locks for the enqueue and dequeue operations. Describe briefly your implementations, and explain how memory management (free list) is done. Is there a possible ABA problem with this implementation? If yes, how is it solved? Is this implementation linearizable? Explain briefly.

Exercise 5 (10 points)

Implement a lock-free, concurrent queue using the compare-and-swap operation (CAS). Describe how you solved the ABA problem. Extend your benchmark to record the number of CAS operations, and the number of failed CAS operations in the enqueue and dequeue operations.

Exercise 6 (20 points)

Benchmark the three concurrent queues, and compare and discuss the results, also against the sequential queue running on one thread.

Run the sequential queue for 1 and 5 seconds per experiment, use batch sizes of 1 and of 1000 values. Repeat the experiment 10 times.

For each of the concurrent queues, run with $p = 1, 2, 8, 10, 20, 32, 45, 64$ threads. Run with batch sizes of 1 and 1000 values per thread. Run the benchmark in four configurations: a) all threads enqueueing and dequeuing with the same batch sizes, b) one thread enqueueing, all other threads dequeueing, c) all threads with id smaller than $p/2$ enqueueing only, the other threads dequeueing only (set batch size for these threads to 0), d) even numbered threads enqueueing, odd numbered threads dequeueing. Perform 10 repetitions of the experiments (or fewer if you have problems with the time).

Plot the results in comparison to the sequential times, and compute/show the throughput speed-up. Analyze your findings.

Exercise 7 (10 points)

A possible application of the lock-free queue is to implement a relaxed queue data structure, where the dequeue (delete) operation can dequeue any element that is in the queue (no ordering constraint). This (useful) data structure is called a concurrent *bag*.

The idea is to let each thread maintain a (concurrent) queue. Insertions per thread are performed using the queues of the threads in a round-robin fashion: pick the next thread and enqueue into this thread's queue (the aim is to ensure balanced queues). Deletions are performed by going through the threads in a round-robin fashion and trying to dequeue from the queue of the next thread. Upon success, the value is returned. If all threads have

been examined and no queue seemed to contain elements (failing dequeue operations for all threads), return failure.

Is this bag implementation linearizable? Try to find point in time for the enqueue and dequeue operations where linearization could happen. The tricky case is a failing delete (dequeue) operation (all queues apparently empty). Give an argument for why this case has a linearization point (where? It may possibly not be some fixed instruction in the code); or give an example showing that a linearization point cannot exist. If not linearizable, is the bag implementation sequentially consistent?

Give brief explanations of your findings and a conclusion on your concurrent bag data structure.

Project Skeleton

To ease the organizational overhead for this project, we provide you with a project skeleton showcasing how to benchmark, organism and plot data gathered from a parallel algorithm. The algorithm itself is not very interesting: its a literal library that lends books to threads.

Benchmarking and data acquisition is done via the python script `benchmark.py` using the `ctypes` [1] python library which provides an interface to call compiled objects directly.

A benchmark now consists of two parts: One part is a function in the python script which takes care of iterating over the number of threads, the specific data assignments, etc. The other part is a function (written in C/C++) in the library which is called by the python script and spawns the given number of threads, assigns the input data, measures the execution time and gathers the performance counters. At the end it returns all the gathered data in a predefined struct to `benchmark.py` for further processing and storage. The exact workings are described in the python docs.

Currently, any benchmark data is timestamped and stored in the `data/` folder where it can be used for plotting later. This has the advantage that gathered data cannot accidentally be deleted, as its immediately written.

Apart from a benchmarking framework you can find all the required OpenMP functions used in `src/library.c`, providing you with a nice starting point for your project!

Anything within the project skeleton may be rewritten by you. We only require certain make targets to exist and run and certain folder structure to be kept, as described below.

What to hand in?

Your *readable* code must be given as a .zip archive generated by

```
make zip
```

Additional source files are to be placed in `src/`. Do not change the folder structure.

We will run your project with the command

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

[1] `https://docs.python.org/3/library/ctypes.html`

to test whether your program compiles, runs and try to reproduce some of your claimed results.

Before submitting test whether the above command runs and succeeds.

You can choose which benchmark you would like to run under the target `small-bench`, but it should be representative for your findings (i.e., not just the benchmark over the library implementation) and should not exceed *1 minute* runtime. Preferably you can present at least one plot for each exercise given. To achieve this in under 1 minute runtime, you may skip averaging over multiple runs (thus save time on not re-running experiments). A subsequent

```
make small-plot
```

should then generate the plots represented by the benchmark under `small-bench`, preferably in a separate pdf document. We will use this to compare your results to the results you present in your report. Code that cannot be compiled on "nebula" with `make` (i.e., make with the 'all' target) cannot get full points.

Your report should consist of short descriptions of your algorithms and implementations as described above and the explanations of and findings from your benchmarking. Algorithms and implementations are to be done in C (or mild C++), and may be described with pseudo-code or actual code-snippets. Mark your report clearly with name(s) and matriculation number(s), and also identify clearly the programs as yours.

Estimated size of the report is around 15-20 pages including all plots and discussion of results. Write the report carefully and correctly and concisely, so that it is readable by an interested researcher. Do not forget to mark clearly with name and matriculation number(s) of the group member(s).

The report should be part of the `.zip` archive uploaded to TUWEL.

References

[1] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, revised 1st edition, 2012.

[2] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, second edition, 2021.

[3] Jesper Larsson Träff. Lectures on parallel computing. arXiv:2407.18795, 2024.