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

Advanced Multiprocessor Programming (AMP)
WS 2024/25
Programming Project

date: 2024-10-28 (October 28)
due date: 2025-01-13 (January 13)

The AMP programming project deals with implementation in C (or mild C++) with OpenMP (or threads) and benchmarking of *concurrent skip lists* as described in the AMP lecture (slides on TUWEL, see also [1, 2, Chapter 14]). You are asked to implement a sequential skip list as baseline for performance comparison and study three concurrent versions of the data structure in different scenarios. The hand-in will consist of a short report on the implementations with a performance analysis including plots or tables 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. 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.

Problem specification:

The skip list is a probabilistic key-value store data structure with expected $O(\log n)$ operation times for lists with n elements. The data structure supports the operations `insert()`, `delete()` and `contains()` on allocated nodes containing an integer key field, a payload as well as pointers and other fields needed to conveniently implement the operations. Following the algorithms from the lecture, the first task is to implement, in C (or mild C++) a sequential version of the skip list to be used as baseline for throughput speed-up comparison. Additionally, it is allowed to compare also to an implementation from a standard library (specify the source!). Based on the sequential version, three (3) concurrent implementations have to be developed. The implementations should be instrumented with performance counters as described in the following. Memory management, making sure that all allocated storage is indeed freed without any risk of memory corruption, is very difficult for this data structure and not required to be implemented. Thus, your implementation may leak memory which of course can make it impossible to run very large benchmarks. 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.

Boiler plate code will be provided, see TUWEL.

Exercise 1 (10 points)

A benchmark for assessing the performance of the skip list implementations shall be implemented. The benchmark runs an experiment a given number of times. An experiment consists of operations on the data structure following a given recipe to be carried out in a given time frame (e.g., 1 second, 10 seconds). It must also be possible to do basic correctness tests. The recipe consists in specifying the mix of operations and the range of keys. It must at least be possible to let all threads select keys from the same range (thereby forcing a large number of conflicting, competing, concurrent operations) and from disjoint ranges (high concurrency, no conflicts).

The following input must be possible in order to set up the benchmark:

- Number of threads.
- Number of repetitions of the experiment.
- Time interval for throughput measurement.
- Number of prefill items
- Mix of operations: percentage of inserts, percentage of deletes, percentage of contains described as (i, d, c) (i: percentage of insertions, d: percentage of deletions, c: percentage of contains).
- Selection strategy: random keys from range, unique (random) keys from range, deterministic (successive) keys from range
- Seed for random number generator (to make the experiments reproducible).
- Base key range.
- Type of key ranges: disjoint , per thread, or common.

Output: Average time per experiment. Total number of operations, number of operations per thread, number of successful insertions (vs. number of insertions), deletions (vs. number of deletions), number of successful contains operations (vs. number of contains).

For correctness, you should make it possible to verify that all items successfully inserted can be deleted once, no more and no less.

Describe your benchmark briefly. Describe anything you did that go beyond the basic requirements (1 page).

Exercise 2 (10 points)

Implement a sequential skip list as outlined above. Describe briefly the node data structure and the implementation ideas at a high level (you may assume everything from the lecture to be known; 1/2-page).

Run the benchmark with one thread. At least document runs with random selection of operations from a base key range of $[0, 100\,000]$ and operation mixes $(10, 10, 80)$ and $(40, 40, 20)$. Run for 1 and 5 seconds.

Find a good way to present the results (plots with number of operations and throughput). Do the number of successful operations match your expectations?

Exercise 3 (5 points)

Make the sequential implementation concurrent by protecting all operations with the same global lock.

Run the benchmark with $p = 1, 2, 4, 8, 10, 20, 40, 64$ threads. At least document runs with random selection of operations from a base key range of $[0, 100\,000]$ both with disjoint and shared range and operation mixes $(10, 10, 80)$ and $(40, 40, 20)$. Run for 1 and 5 seconds.

Plot the results in comparison to the sequential times (throughput speed-up). Comment on (analyze) your findings.

Exercise 4 (15 points)

Implement a concurrent lazy skip list with fine grained locking. Describe the main implementation ideas (1/2-page).

Run the benchmark as in the previous exercise with $p = 1, 2, 4, 8, 10, 20, 40, 64$ threads. At least document runs with random selection of operations from a base key range of $[0, 100\,000]$ both with disjoint and shared range, and operation mixes $(10, 10, 80)$ and $(40, 40, 20)$. Run for 1 and 5 seconds.

Plot the results in comparison to the sequential times (throughput speed-up). Comment on (analyze) your findings.

Exercise 5 (20 points)

Implement a lock-free skip list. Describe the main implementation ideas (1/2-page).

Run the benchmark as in the previous exercise with $p = 1, 2, 4, 8, 10, 20, 40, 64$ threads. At least document runs with random selection of operations from a base key range of $[0, 100\,000]$ both with disjoint and shared range and operation mixes $(10, 10, 80)$ and $(40, 40, 20)$. Run for 1 and 5 seconds.

Plot the results in comparison to the sequential times (throughput speed-up). Comment on (analyze) your findings.

Exercise 6 (10 points)

It is possible to get bonus points for the project by for instance (not exhaustive, and not inclusive):

- Doing a more thorough experimental evaluation
- Doing an overhead analysis with performance counters: how often are the lock-free retry loops repeated? How often do compare-and-swap operations fail? How often is a list retraversed?

- Implementing ideas for early linearization as discussed in the lecture [4], avoiding (as far as possible) retraversal of (parts of) the data structure (is helping excessive?).

What to hand in?

Your code (which should be readable) is an important part of the hand-in, and must be given as a .zip archive.

We will run your project with the command

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

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

[4] Jesper Larsson Träff and Manuel Pöter. A more pragmatic implementation of the lock-free, ordered, linked list. In *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 457–459. ACM, 2021.