Exercise 4 — 20P

submission date: **9. Jan (23:59)**

This exercise includes a number of data files. You can download them with the included script, or manually from `https://algo2.iti.kit.edu/download/efficient-parallel-cpp-ex4.zip`. Use the provided classes to read them into your program.

Important: Do not add the data files to your repository, or you will quickly exceed your quota.

4.1 Producer Consumer Data Structures (8)

For the beginning of this exercise, we want you to construct a concurrent queue (FIFO-Container) that stores elements in a "circular array", by using atomics, the goal will be to achieve a low contention on those atomic operations. Use a constant-sized array with n slots to store the created elements ($n >> p$). The element type should be templated, but you can assume `std::atomic<T>` to be specialized (lockless) and you can assume that the element given by the standard constructor (`T()`) will never be inserted. Therefore, it can function as a dummy element (denoting an empty slot). For the second part of this exercise, we want you to implement an arbitrary container that stores elements. The constraints are basically the same as before (`std::atomic` is lock-free and `T()` can be a dummy) but the order of returned elements can be arbitrary, it does not have to correspond to the insertion order.

a) Implementation (3)

Implement the concurrent FIFO-queue, paying special attention to the critical parts of the execution (when the data structure is full or empty). Assume that there is always at least one writer and one reader, therefore waiting on an element to extract (when the structure is empty) is fine, and waiting for an empty spot (when it is full) is also fine.

b) Implementation (2)

Design and implement another arbitrary concurrent data structure, comparable to your queue, how can you improve your data structure, when you don't have to maintain the element order (also under the same assumptions as before).

c) Microbenchmark (1)

Write a quick micro-benchmark that uses `std::thread`s to tests your data structures. You should have a number of inserting threads, and a number of consuming threads. Each thread should execute nothing except repeat its task until a certain number of elements has been handled.

d) Evaluation (2)

Show your results with a varying number of producing and consuming threads ($p_+ + p_- = p_{all}$).

4.2 Work Tree Scheduling (7)

In this exercise you should use the previously implemented containers to implement a normal work scheduling scenario. In the framework for this question you can find a class representing a work tree. You can use the constructor argument to scale the work and speed up or slow down your experiments (try numbers between 0 and 1). Explicitly also try scale factors close to 0 (for shorter tasks the scheduling overhead becomes more important).

Executing a task will give you a varying amount of new tasks.

a) Naive Algorithm (1)

Implement the naive work scheduling technique that inserts each new task into the concurrent work queue. For this to work, you should allocate the queue large enough that it is impossible that all threads are waiting for an empty spot (full structure with no reader). Additionally, you should think about what happens at the end of the execution; make sure that not all threads are waiting for new jobs (without a producer). Maybe insert some end of computation tasks that make the thread that is executing them stop working.

b) Optimized Work Scheduling (3)

Now in this task, we want you to be somewhat smarter about the scheduling. You should use a local queues (maybe even non-concurrent) per

thread. This will reduce the contention and thus the running time of accesses. There still has to be some global scheduling mechanism, that is used to redistribute tasks among cores.

One option could be to have one non-concurrent queue per thread, and also one concurrent global queue. Then you would only have to push some elements into the global queue (enough to balance work imbalances).

Another option could be to have the local queues concurrent and to access another threads local queue, when there is no local work.

Note: You only have to implement one of those options, but maybe vary how tasks are "communicated" between threads (which tasks are pushed to the global queue/which local queues are accessed).

c) Evaluation (3)

Vary different parameters within your implementation, such as thread count, task communication etc.

Also measure at least one non-standard metric (not running time based). This metric should deliver some insight into the work balance or the work stealing/work communication.

Note: Also see what happens if you scale the work done by each task between 0 and 1 (constructor parameter for the tree class).

4.3 DAG Scheduling (5)

DAG (Directed Acyclic Graph) scheduling is a somewhat more difficult problem. Each task can be seen as a node within a graph. Each incoming edge symbolizes a task that the current node is dependent on. Each task can only be scheduled once all its dependencies are fulfilled.

a) Scheduling (3)

Implement a scheduling algorithm that pays respect to the described constraints.

b) Evaluation (2)

Vary different parameters within your implementation, such as thread count, queue sizes, scheduling strategy etc.

Note: Also see what happens if you scale the work done by each task between 0 and 1 (constructor parameter for the DAG class).