Exam Assignments 4

Parallelizing Divide and Conquer Algorithms with Tasks

Divide and conquer algorithms solve a problem by recursively splitting it into smaller pieces and solving those.

A way to parallelize these types of algorithms with OpenMP is the `task` construct. Tasks are independent units of work that can be dispatched at run time. They then get assigned to threads by a dynamic queueing system, where they are executed in parallel.

Using tasks, a divide and conquer algorithm can be parallelized by creating new tasks everytime the algorithm splits up the problem, and assigning each subproblem to one task. It should be noted that a `taskwait` should be inserted when the result of one tasks computation is needed for the following code.

Speeding up Merge Sort

Parallelizing

A big improvement in performance can be achieved by parallelizing the first part of the algorithm using `tasks` in the way described above.

Different Algorithm for small n

Although merge sort has a good time complexity, allowing it to be faster than other algorithms for large inputs, it can be beaten by simpler algorithms for small n. Thus, something like insertion sort can be used for small n to speed up the computation.

Use Stack instead of Heap for smaller n

Dynamically allocating memory on the heap takes some time, it's quite a bit slower than using memory on the stack. The algorithm can be sped up a bit by allocating the array that the subarrays get merged into on the stack when doing it on the heap for smaller n, when doing it on the heap isn't worth it.

Avoid copying Data

Each time we merge we create a new array, when we could be reusing one array each time. By creating two merge sort functions that alternate between each other, one merging into a buffer array, the other merging into the original array, we can avoid allocating new memory altogther.

Parallelizing the Merging

We can speed up the merging step using the technique described in the following section.

Multithreaded Merging

Multithreaded merging is a recursive divide and conquer algorithm that merges two sorted arrays. It is called "multithreaded" because, in contrast the usual merging algorithm used in merge sort, it can be parallelized (using, for example, `tasks`).

The idea is to split both of the arrays into two (nearly) equally large halves using the median of one of the arrays as the pivot value in both of them. Then we recursively merge the two lower halves and the higher halves while placing the median between them.

The placing of the median is the actual work being done, the algorithm is done when each of the array elements was the median in one of the computations.

Interesting Things in "What every systems programmer should know about concurrency"

1. Compiler optimizations and sequential consistency

Before I read the text, I already knew that compilers make a host of optimizations to the code I write, but I didn't know that they would even shuffle the order of my statements around if they saw some performance gain to be had there.

Learning this and the steps a programmer has to take to prevent the compiler from fiddling with their code in concurrent programs was fun!

In particular, the idea of sequential consistency and the way it is implemented on **weakly ordered** hardware (didn't know this existed) like ARM with memory barriers, aswell as the way even this concept (sequential consistency) can be softened for performance using different memory orderings.

2. Spurious LL/SC failures

When answering the question about `std::atomic::compare_exchange_weak` in the second exam assignments, I also found out about `compare_exchange_strong` and the fact that `compare_exchange_weak` can spuriously fail. I was wondering how and why it would do that, so I'm glad I could find out about that here.

On many RISC architectures, there are no dedicated cpu instructions for reading, modifying and writing atomically. Instead, they implement atomic operations using the instructions `LL` (**load-link**), which loads from an address and instructs the processor to monitor it, and `SC` (**store-conditional**), which stores to an address if it hasn't been written to since the `LL`.

However, monitoring addresses on the byte level is too resource intensive, so it's instead often done on the level of cache-lines. Thus, if a different address on the same cache line as a linked address is modified, the processor will mark the whole cache line as modified. This causes the next `SC` (and in turn `compare_exchange_weak`) to fail even though the address itself wasn't ever written to.