CPEG655

Lab 03

Michael Hutts

Problem 1: Synchronization

Task 1

Upon initially writing the functions I had an error when compiling of the due to a conflict with the name "remove". For this reason note I've renamed "remove" to "delete".

Going about implementing the functions was relatively simple. For all functions I created a pretty classic recursive functions for traversing a binary tree. I was a little nervous of how recursion might interact with threading/locking down the road, but it seemed like the ideal method performance-wise so I left it to be. All initial task 1 code can be seen within "synchronization.c"

Task 2

The workload for Task 2 was implemented as described.

Task 2.1

For Task 2.1, I defined a simple pthread_mutex_t variable to lock accessing the tree when it is being accessed by a thread. With this implementation I could simply call pthread_mutex_lock() before running any of the functions from Task 1 inside the workload. This was the simplest thread-safe implementation I could think of, but I noticed a substantial slowdown when using N=64. I initially profiled the software with PAPI using PAPI_TOT_CYC as my metric, but this did not offer much context.

synchronization_thread.c (PAPI Profiling)

N	**64**	**1048576**
PAPI_TOT_CYC	184478.3	173561.5

It immediately became clear to me that this was a poor primary metric to profile our multithreaded program, and we should instead be looking at runtime explicitly as a metric.

synchronization_thread.c (time.h Profiling)

N	**64**	**1048576**
Seconds	40.044	33.44404

Immediately the performance disparity between values of N became clear. As locking the entire tree means that we won't see much performance difference in relation to locking, this performance difference can be contextualized as a purely a matter of the structure of the data and its performance impact. My best speculation on the reasoning for this would be that the wider range of N would offer a "wider" tree compared to one without duplicates. Originally I had written add() to throw away duplicate values, and there I saw a reversed relation performance wise, as 64 would of course produce a far shorter tree.

Task 2.2

This is where a roadblock in the recursive structure of my code was found, and I was never truly able to *fully* overcome it. Creating the fine-grained locking mechanism was incredibly simple: just add a lock to each node. The problem came when attempting to adapt the existing functions to this locking approach. The simple initial implementation worked completely with the new locking but at a steep performance impact. My initial approach was to lock a node, recursively call the function, and unlock that node. The problem was that this in practice ends up locking the whole tree. Working around this in the same structure might be possible if the functions were reworked to take parent nodes and then unlock those nodes once entering the child, but this seemed against the principal of the original prompt. Instead, the simplest solution seemed to be to abandon the recursive function and switch to a while-loop instead. This went well for add(), which did seem to have a minor performance uplift from locking only the active nodes instead of just the entire tree. The problem came with remove(), which was much more complex to transition away from a recursive structure. The problem was that a much larger amount of data was needed to be locked to transform the tree in a thread-safe manner. This led to branches of the tree unintentionally getting broken off in the process of multiple concurrent transformations. Unfortunately I was unable to get this full thread-safe, optimized solution working. I strongly believe that if the source of the errors could be squashed, this would perform far better than the Task 2.1. For this reason I've included my fine-grain optimized code for remove() in *synchronization_threaded_fine_grained_full.c*, but could not profile this accurately due to the issues I experienced.

Despite these setbacks, it's still worth reviewing the data and speculating on the matter if the performance penalty of remove() hadn't been so severe. As explained previously, the major benefit of fine-grained locking is the ability to free the tree as we traverse it. As long as destinations of the different threads don't overlap, they can work completely concurrently with 0 idling. In the case of remove() it got all overhead of a more granular approach, with none of the performance uplift. Regardless, some performance characteristics can still be drawn from the profiling.

synchronization_thread_fine_grained.c (time.h Profiling)

N	**<u>64</u>**	**<u>1048576</u>**
Seconds	71.342	70.020

In comparison to Task 2.1, no performance was seen between N=64 and **N=**1048576. This is most likely to the better performing add() removing any performance disparity between these two. As to how a fully fine-grained approach might help for each individual case, N=64 would see the most uplift in being able to quickly free repeated paths as it goes. With a low value of N and multiple paths, the range of potential values and therefore tree paths and diminished, so the sooner each node in that path can be unlocked, the faster the next thread can traverse it. In comparison a high value of N gives a substantial amount of potential paths to traverse for each thread, and low likelihood of repeating any one value. This has an extreme all-around benefit compared to a full-tree lock, where each value can now traverse its own individual path without different values ever needing to wait on a lock to traverse an unrelated path.

Compilation and Testing

To compile and run the code, simply do *synchronization* synchronization*.c -lpapi -lpthread*, where * is the specific version being tested (_threaded, _thread_fine_grained, etc.). For section 2, the specific values of N are defined on the line *const int N = 64; // 64, 1048576*. Simply change out the value and recompile to compare. Runtime comparisons are seen in threaded_*_clock_*.txt and PAPI results in threaded_*_*.txt. Integrity checks are left out of the file by default due to printf() throwing off profiling, but can be enabled by simply uncommenting their lines within workload().