Project Summary: Implementation of a Work-Stealing ThreadPool

Project Overview

This project primarily focused on the creation of a work-stealing thread pool for the parallel execution of divide-and-conquer algorithms, utilizing a thread pool implementation for dynamic task parallelism. The aim was to prevent excessive resource use, hence avoiding crashes or deadlocks. As an alternative, a work-sharing approach was also considered, albeit for reduced credit.

Approach and Implementation

During the execution, I implemented two main approaches of multi-threading, which were work sharing and work stealing. I discovered that work stealing led to better load balancing and reduced synchronization requirements, as it reduced contention by maintaining individual queues of tasks for each worker thread. The tasks were added to the bottom of a worker's queue, creating a last-in, first-out (LIFO) order of execution. If a worker ran out of tasks, they could check a global submission queue or attempt to steal tasks from the top of other workers' queues.

Handling Future Tasks

To avoid thread starvation, when a worker attempted to resolve a future task not yet computed, it was designed to either steal and execute the task itself or aid in executing tasks spawned by the task being joined. The project assumed a fully-strict model, meaning each task submitted would have a matching call to future get() within the same function invocation. All tasks spawned by a task were considered subtasks that need to complete before the task completes.

Synchronization and Signalling Strategies

In regards to implementation, I had complete freedom, so long as I adhered to the constraints imposed by the API and resource availability. I had to design a synchronization strategy to protect data structures such as task execution flags, the local queues, and the global submission queue, and I had to develop a signaling strategy so worker threads would learn about task availability.

Defining Structures and Implementing Functions

Structures for a future and a thread pool were defined in threadpool.c. The future structure stored a pointer to the function to be called, any data to be passed to that function, and the result when available. The thread pool kept track of a global submission queue and the worker threads it started. I also implemented numerous functions, such as thread_pool_submit(), future_get(), thread_pool_shutdown_and_destroy(), and future_free() to manage task submission, completion, shutdown of the thread pool, and memory management respectively.

Optimizations and Extensions

In the second part of the project, I considered and researched optimizations to minimize per-task synchronization overhead. Specifically, the THE protocol and Chase and Lev's work-stealing deque offered valuable insights. The project also discussed extensions to support computations that are not fully-strict but still recursive, which could possibly lead to a deadlock situation.

Finally, the project considered extensions to support fully general acyclic computational graphs. This required an inverted control flow model, similar to that used in Java's CountedCompleter classes. Thus, tasks were not joined, but rather a callback would be executed once they completed, allowing dependent tasks to be scheduled.

Conclusion

In conclusion, this project not only provided me with valuable knowledge about thread pools, task parallelism, and synchronization strategies but also allowed me to experiment with different implementation strategies and optimizations. It also offered an understanding of potential deadlock situations and how to avoid them, and finally, it pushed me to consider how to support a more complex, fully general acyclic computational graph.