Thread pool

Bug Lee (bug19)
Dana Altarace (daltarace)

Introduction

Thread pool is a powerful abstraction that provides a safe and efficient environment for parallel programming. By using a thread pool, excessive resource usage can be avoided which can lead to unexpected crashes during the execution of a program. For our thread pool implementation, three strategies were used to better optimize the performance. Firstly, the work-stealing approach was used, instead of work-sharing, to avoid the situation where the global deque becomes a point of contention as workers submit smaller internal tasks. Secondly, we have reduced the workers' sleeping time in the future_get() function by applying the work helping approach. Lastly, additional locks were added to minimize the unnecessary blocking time that cause by the overlapping lock acquisition over the same lock among workers. We have made each worker thread have their own lock to protect share access to individual local deque and the thread pool itself own a separate lock to protect share access to the global deque.

Main thread

The thread pool can be created using the thread_pool_new() function where initialization of the global deque, lock, conditional variable, barrier, shut down flag, and creation of a given number of new worker threads happen inside this function. A barrier was needed for synchronizing the thread pool and all workers in the beginning. It prevents problematic situations such as workers trying to access other workers' local deque where the other workers are not set up yet. Once the thread pool is created, work can be submitted via the thread_pool_submit() function externally. It forms a future using the task function pointer and its arguments given from the parameters, then submits the new future to the global queue (external submission). After the future has been submitted, it will wake up any worker who is sleeping to inform that there is work do to. If by the time when future_get() function get called but the submitted work has not been completed yet, the main thread has nothing to do and wait for the submitted work to be completed by the workers. Once the work is done, it frees the resource held by the corresponding future. In the last step, it free all the resources held by the thread pool through the thread_pool_shutdown_and_destroy() function. First, it set the shutdown flag and broadcast to all workers to wake up. Then, it waits for all the workers to come home (join) and release any of their resources once they join back. Finally, it releases the remaining resource from the thread pool.

Worker thread

Worker threads are created during the thread_pool_new() function and wait until all other co-workers come to work by use of barrier (discussed above in the main thread section). Once all workers are ready to work, the worker thread enters the infinite while loop so that the worker can work continuously until the shutdown flag is set by the thread pool. A worker checks if there is any work to steal from co-workers by checking whether their local deque is non-empty. If not, then it checks for the shutdown flag from the thread pool so that it knows when to stop working and go back home. If none of the previously described conditions are met, the worker goes to sleep until someone wakes him up. On the other hand, if there is work to do, the worker either fetch the work from the global deque or steal (discuss in the Work stealing section) the work/future from the other workers. In our fully-strict workloads model with the addition of work helping, it was safe to eliminate the case for the worker fetching work from its own deque. After the execution of the fetched work, the work gets marked as completed and awake main thread or workers that are waiting for that work to be done. In most case, dequeued work generate a sub-task during the execution by calling thread_pool_submit() internally. In that case, it forms a future that wraps the function pointer and arguments, then puts the new future to the top of the current worker's deque. It also informs any workers that there is work to steal. Soon after, it will be followed by future_get() since thread_pool_submit() and future_get() come in pair. If the submitted sub-task is not yet completed when it reaches the future_get() function, the current worker tries to help finish the submitted sub-task or wait for it to be completed by other workers (discussed in the Work helping section). Once the submitted sub-task gets completed, the worker releases the resource held by the corresponding future.

Work stealing

The current worker finds the first non-empty local deque from other workers by iterating through workers array inside the thread pool. If it finds one, then it fetches the last element of the identified deque and starts the execution of the fetched work. Once the fetched work is done, it marks the fetched work as completed and wakes up any of the workers that are waiting for that work to be completed. This ensures that the workload of each worker is balanced, and we observed several magnitudes of speed up in the test program as a result.

Work helping

In order to simplify our design to a fully-strict workload model, a basic form of work helping was needed. When the current worker calls the future_get(), it will 'steal' the given work from its own deque if the work has not been stolen by the other workers. Otherwise, if the given work got stolen by other workers, the current worker simply wait for the other worker to finish the given work. Work helping was mainly needed to simplify the work-checking strategy for the worker thread, which was helpful for reducing the implementation complexity of the multi-lock design.

Multi-lock design

As mentioned in the introduction, we assigned one lock to the thread pool and one lock per worker. The goal of this design was to make the Fibonacci test program run faster since workers were mainly blocked due to lock acquisition for the single lock design. The goal of this design was to spread out the resource related to lock acquisition more evenly but also ensure that the critical shared data does not get accessed by multiple workers at the same time. Here are the parts that involve multi-locking.

- **steal_work() function**

> There are two things to note in this function. First, the global pool lock gets released at the beginning of the function and re-acquire at the end before returning. This is to avoid double locking. Another more important thing to note is that, once the current worker acquires the lock from the owner of the local deque and steals the work, it is guaranteed to be removed from the original owner of that work when the lock gets released. This should be true as long as other workers can only access local deque by acquiring a lock from the owner of that deque.

- **Accessing global deque or shut down flag**

> The current worker simply acquires global pool lock anytime it needs access to the globally shared data among workers.

- **thread_pool_submit() function**

> Each thread (either main or worker) acquire their own lock before submission to their own deque, then releases the lock after the submission. During the submission, it also defines the owner of the work as a current worker who is submitting work.

- **future_get() function**

> For the main thread, it acquires the global pool lock and releases it when it is sleeping or once the work is completed. For the worker threads, it acquires the lock from the owner of the work (who owns the local deque that contains this work). This prevent any potential conflict with thread_pool_submit() and/or steal_work() functions. If steal_work() acquire the lock first, then the work is guaranteed to be running or completed once future_get() acquires the lock. If future_get() acquire the lock first, then the work is guaranteed to be removed from the owner's deque (or already have been removed from the deque if it has been started by other workers) before releasing the lock.

Others

Padding was added to each data structure to minimize false sharing inside the cache. However, no significant performance gain was observed.

Valgrind/Helgrind output

The only error we observed from the valgrind/helgrind run is the 'dubious lock error'. This is caused by intentionally leaving the pthread_cond_signal(&pool->workAvail) outside of the global pool lock. This was done so to reduce the effect of true sharing in the cache. In addition, this is not an issue for our implementation since the order of this signal does not matter. If the signal got sent before any workers were sleeping, steal_work() should still see that there is work to steal. If the signal got sent after any workers were asleep, then it simply wake up the worker, and steal_work() should see that new work has been submitted.