Thread Pool Implementation in C

Jialin Li and Edward Hu

***Abstract*—This is a short report on our thread pool performance.**

I. Introduction

THREAD is a basic unit of CPU utilization, it consists of program counter, register set, and its own stack. Compared to processes, threads are much more lighweight since virtual memory space remains the same during a thread switch, which is not true for process switch. However, even threads sometimes pose overhead that will affect the performance of the program. Initializing and cleaning threads can pose a certain amount of overhead. For an parallel application that creates and destroys a large number of threads that each only runs for a short time, the thread management mechanism can create unnecessary resource waste. Thread pool can be useful in this type of scenario as it reduces the complexity of thread management and the overhead involved in thread creation and cleaning. The thread pool in implement in **GNU C**.

II. Motivation

We want to be able to implement a thread pool that is convinient to use. We aim to create a provide an interface that is easy for programmers to use. Thus we minimize the number of functions exposed to the programmers to maintain simplicity. We also want the performance of the thread pool to be reasonably more efficient than normal thread-based version. Thus we create several test cases to compare the performances in different implementations. We mainly focused on testing the performance difference in thread-based implementation and Goroutine-based version for certain types of jobs. Finally, we simply want to learn how to implement a thread pool and explore the drawbacks along with possible improvmetns.

There are several goals we are aiming to achieve in the experiment. First, we want to run both long and short tasks in the threal pool to analyze which type of tests have the most impact on the performance on the program. We've created created various tests for the thread pool to perform and we will show the data collected later in the analysis section. We hope the low overhead of switch between tasks will help us outperform thread-based implementation as well go-based implementation. In addition, we implemented work stealing mechanism to improve the work throughput done by the thread pool. We hope the work stealing function will further improve the efficiency of the thread pool.

III. Programming Interface

The programming interface is designed for simplicity, there are three functions users are exposed to.

- `void thread_pool_init(int workers, int mutex_flag);`

  This function initialize a new thread pool. The **workers** attribute specifies the number of threads that can exist in a thread pool. If 0 is passd in, the number of threads will be initialized to the number of core in the execution environment. The **mutex_flag** argument let user decides whether to use mutex or spinlock as synchronization method. Note if the thread pool is initialized more than once, every invocation of this function will be discarded.

- `bool thread_pool_add(task_func *func, void* aux, enum CallType);`

  This function adds tasks to the thread pool. The **func** attribute represents the function that is to be executed. The **aux** attribute indicates the parameter **func** takes in. the **CallType** specifies whether to use blocking or non-blocking mechanism for the thread pool. If a task is added without failure, $true$ is returned. Otherwise, $false$ will be returned.

- `void thread_pool_wait();`

  This function signifies all threads in thread pool that no more job will be added and the calling function waits for all tasks in the thread pool to finish.

IV. Implementation

The low overhead of the thread pool is acheived with several design decisions. We will discuss the major implementations. A thread pool is first initialized by the main thread and the corresponding number of threads are initialized with their attributes. Each task is represented as a **struct task**. It is created by the calling function whenever the user sends the function to be executed to the thread pool. It is then distributed to the threads in the thread pool in a round-robin style to ensure even distribution. Each thread owns a privat task queue to store all its assigned tasks. Each queue is protected by either a mutex or a spinlock depending on users' specification. However, it is possible that a thread's task queue is empty when it is first initialized. It will waste CPU resource if such task is kept up and running without executing any tasks. Thus, we use semaphore to improve the efficiency of the thread pool. Thus we let the main thread invoking sema_post on each thread once it finishes adding tasks to the thread's task queue to indicate the availability of tasks. However, the thread's task queue can be empty for other reasons. The most important one is that the thread finishes executing all

assigned tasks. Thus, we implement work stealing algorithm to keep the idle thread busy by letting it trying to steal tasks from other threads by randomly choosing one. Since we don't want tasks stealing to create too much overhead as it tries to grab tasks from other threads, we simply let the idle thread using pthread_mutex_trylock() or pthread_spin_trylock(). If the stealing succeeds, the tasks is grabbed from the victim's task queue and executed. Otherwise, the steal will fail. If the steal fails too many times, that indicates many threads' task queue is already empty and the thread pool is getting close to running out of tasks. In this case, letting more threads trying to steal tasks will create unnecessary overhead, thus the thread will be blocked to prevent wasting resources. If eventually the threads run out of tasks, it will increase **wait_sema** to let the main thread know it finishes its job. If all threads signify that they've finihsed their tasks, then thread pool is terminated.

V. Analysis

Below is the hardware we used to conduct experiements on the performance of the thread pool.

TABLE I
HARDWARE SEPCIFICATION

Processor	Intel Core i7-6700 CPU @3.40GHz × 8
RAM	15.6 GiB
OS	Ubuntu 16.04.4 LTS 64-bit
Compiler	GCC 5.40

We create same tasks for every implementation to execute to ensure fairness. we've selected three types of tests that represent three typical scenarios in most applications. We designed a task that accesses file system and performs read and write operations. Since accessing file system usually requires blocking which generates large overhead, this task is perfect to represent tasks that takes more than a few CPU cycles to finish. We also designed a task that task an argument, and then computer then sum based on the input and generates a simple hash value. This represents a short to medium tasks present in most applications and we selected it as a candidate to represent short tasks. We also include a task that does nothing in order to test sheer performance of the thread pool without the influence of any outside factors. In addition, we mixed the number of long and shorts tasks in each benchmark in order to generalize the real life scenario where a mixture of different tasks are executed. We want to show that the speedup is not or at least minimally affected by the type of tasks executed. Then, we test the scalability of the thread pool itself by gradually increase the number of threads in the thread pool running $10,000$ tasks. The result matches with our expectation as the scalability is achieved when increasing the number of threads in the thread pool.

VI. Conclusion

Theoratically, thread pool can be very efficient especially dealing with a large number of relatively short tasks concurrently. In order to acheive maximum performance

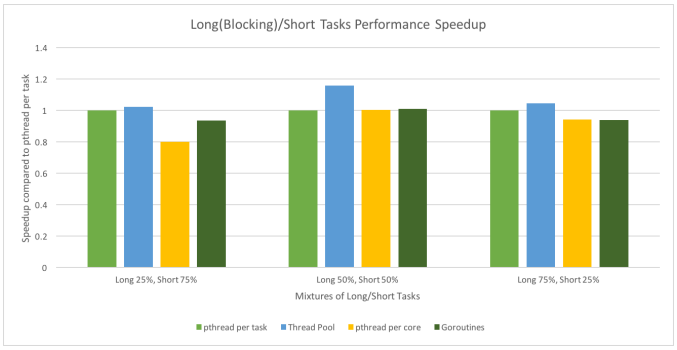

Fig. 1. Benchmark speedup using long blocking tasks and short tasks. From left to right each section represents: long 25% with short 75%, long 50% with short 50%, and long 75% with short 25%. Green bar: pthread per task. Blue: Thread pool. Yellow: pthread per core. Dark Green: Goroutines

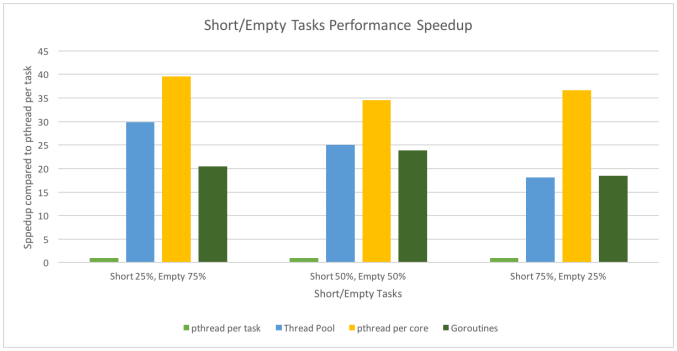

Fig. 2. Benchmark speedup using short blocking tasks and empty tasks. From left to right each section represents: long 25% with short 75%, long 50% with short 50%, and long 75% with short 25%. Green bar: pthread per task. Blue: Thread pool. Yellow: pthread per core. Dark Green: Goroutines

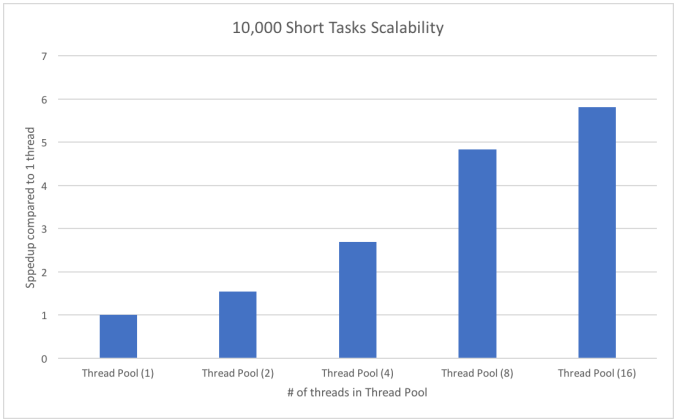

Fig. 3. Benchmark scalability of thread pool with $10,000$ tasks running. Threads used from left to right: 1, 2, 4, 8, 16.

boost, the use of context switch should be minimized. The thread pool does a greate job of reducing the overhead in context switch. In addition, work stealing further facilitates the work throughput by letting idle threads execute other threads' tasks. However, thread pool is not a silver bullet that solves every problem of thread. The overhead of thread context switch is transfered to each thread's task queue as executing tasks requires dequeing the task each time, which

also applies to job stealing. We've also conducted experiement with creating only one task queue shared between all threads, but the same thing still applies here that the overhead is transfered to the queue since accessing tasks is performed with lock's protection. In addition, round-robin style task distribution may not be the best way to distribute tasks as calling thread is unware of the actual workload in its distribution target. The work stealing is also very tricky to implement as the stealer is not aware of the workload distribution during runtime. If the majority of threads finish execution while a few survivor threads with many tasks are still alive, the parallelism can be compromised since a few threads become the bottleneck that causes main threads waiting unnecessarily. Additional improvements such that runtime dynamic workload check and compiler support for user-guided optimization will further increase the usibility of thread pool.

In summary, the thread pool is great for running a large amount of tasks effciently with relatively small number of threads. In parallel applications with many short tasks to perform thread will be a strong candidate solution.