JEThread: A Thread Pool Implementation in C

Jialin Li and Edward Hu

***Abstract*—JEThread is a Thread Pool written in C. It provides an easy interface and good performance for concurrent programming in C. JEThread achieves good performance across a mixture of workloads and automatically load balance through techniques like work stealing.**

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 lightweight since virtual memory space remains the same during a thread switch, which is not the case for process switch. However, even threads sometimes pose overhead that affects the performance of the program. Initializing and cleaning threads is one of the biggest contributors to the overhead. For a concurrent application that creates and destroys a large number of threads that each only runs for a short time, the thread management mechanism can cause 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 destruction.

II. Background

Without a thread pool, the common techniques used to do concurrent programming in C are to spawn a thread for each task if it is long running, or to create a fixed number of worker threads and statically partition the work among the workers manually. Although both ways are feasible and provide control freedom to programmers, they can also add unnecessary burdens and require programmers to have a great deal of knowledge of both the workloads they are running and the type of thread programming model that would result in good performance. Newer languages like Golang deals with this problem by providing Goroutines which are extremely lightweight user threads, as well as other concurrent native objects to better support concurrent programming. These features have brought Golang many users. For example, container engines like Docker and LXD, container orchestration tools like Kubernetes are all written in Golang. While it is hard to fully integrate runtime concurrent programming support in C, the thread pool abstraction provides a good middle ground by providing easy programming model and efficency brought by C.

III. Motivation

Our primary goal is to implement a thread pool that is easy to use and flexible with different workloads. We minimizes the number of functions in the thread pool interface to keep it narrow yet fully functional. We also make a distinction between blocking and nonblocking tasks, and adopt techniques like work stealing to support various workloads.
We also want the thread pool to be reasonably more performant than previously mentioned pthread-based approaches. Thus we create several test cases to compare the performance in different implementations. Finally, we simply want to learn the inner-working of an efficient thread pool and explore the drawbacks along with possible improvements.

IV. 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 initializes 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 cores 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 initialization after the first one will sinply 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 to thread pool without failure, $true$ is returned. Otherwise, $false$ will be returned.

- ```
  void thread_pool_wait();
  ```
  This function blocks until all threads in thread pool have finished all existing jobs.

V. Implementation

The low overhead of the thread pool is achieved with several design decisions. We will discuss the major implementations.
  ```

A. Task Assignment

Each task is represented as a **struct task**. It is created by the calling function whenever the user sends a function to be executed to the thread pool and then the task is distributed to the threads in the thread pool in a round-robin style to ensure even distribution. We keep track of the current worker that we are assigning to, and advance it after we've assigned N tasks to it. This is to reduce the overhead of waking up workers when the number of tasks is few, and to try batching tasks and provide better tasks locality. This is based on the hypothesis that tasks created close in time are likely to reference similar memory regions. Therefore, we hope to keep them on the same worker thread. Even when this hypothesis does not hold, it does not hurt JEThread's performance. When a job is assigned to a worker, the main thread grabs the lock(spinlock suggested) to the worker's private queue and add it to the queue, then it signifies its target that new tasks are available by incrementing the target's semaphore.

B. Blocking versus Non-Blocking Tasks

In addition to batch assigning jobs, we also treat blocking and nonblocking tasks differently. This is necessary because JEThread by default starts with N threads, where N is the number of processors, and if all worker threads are executing some blocking tasks, then nonblocking tasks will not be able to execute until much later. It becomes even worse when the majority of tasks are blocking tasks. In such case, the fixed number of workers would be blocked constantly. To solve this problem, we use an approach similar to Golang's parking threads, where we create a new thread to handle blocking tasks and only use our default set of worker threads to handle nonblocking tasks. In order to implement this, we need to identify if a task is blocking or nonblocking. This is difficult and inefficient to do at runtime, but can be done much easier in compile time. In our implementation, we rely on user to specify the CallType when adding a task. This is not fundamentally necessary - we could have modified GCC compiler or inspected task at preprocessing time to accomplish the same thing, but due to the time constraint we exposed this option to users. We believe that users would try their best to provide accurate CallType because they want better performance. Even if they accidentally misidentified CallType, this would not effect the correctness of JEThread.

C. Work Stealing

One of our goals is to achieve automatic load balancing. For this purpose, we implement work stealing for each worker thread. For each worker, we first do a sem_trywait() to see if there is any job on its private queue. If there is none, then it randomly selects a victim thread. Since we don't want work stealing to create too much overhead as it tries to grab tasks from other threads, we first let the stealer try peeking at the victim's queue using pthread_mutex_trylock() or pthread_spin_trylock(). If the stealing succeeds, a task 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.

This is our first attempt at work stealing. To optimize this, we could have kept randomly selecting victims until there is a victim with work above a certain threshold or up to a certain number of tries. To avoid all workers trying to steal when there is no tasks available, we can also use randomization to determine if a worker should try work stealing or just block until it is assigned a job. Another optimization we can perform is to steal more than one work at a time since we want it to be worth the overhead of running the algorithm. Although not reflected in the evaluation, we did try to evaluate the impact of the current work stealing techniques and found that even the naive approach improves JEThread's performance slightly.

D. Internal Synchronization

Each worker thread owns a private task queue to store all its assigned tasks. Each queue is protected by either a mutex or a spinlock depending on users' specification. We highly recommend using a spinlock because we've ran experiments for both types of locks and observed that the spinlock always provides better performance. This is because the critical section is small and the code design rarely leads to lock contention.

In addition to preserving correctness through protected access to each worker's queue, we also try to keep JEThread's overhead low through efficient implementation. For example, when a worker runs out of tasks to execute, it can either busy wait or block. Because busy waiting will exhaust CPU resource, we decide to use a semaphore **task_sema** to block a worker while there is no available task. When the main thread assigns tasks to the worker thread, it calls sema_post on the worker's task_sema.

In order to wait for all worker threads to finish, in thread_pool_wait() we first mark each worker's private state **count** to true, then wake up each worker using **task_sema** , and wait on each worker's **wait_sema**. Once a worker wakes up, if there is no more job on its private queue and **count** is true, it wakes up the main thread to indicate that it has finished the current set of tasks.

VI. Evaluation

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

There are several goals we are aiming to achieve in the experiment. Firstly, we test the flexibility of the thread pool by running a mixture of long(blocking) and short tasks and evaluate how we perform compared to other implementations. Secondly, we evaluate the overhead of JEThread by running a mixture of short and empty tasks. The tasks we ran are common tasks in most web servers. The long task accesses file system, creates a file and writes 1106 bytes to the file. Since accessing file system usually requires blocking which generates large overhead, this task is perfect to represent tasks that takes lots of CPU cycles to finish. The short task generates a simple hash value and the empty task does nothing in order to test sheer overhead of the thread pool without the influence of any outside factors.

To show flexibility, we show that JEThread is able to outperform at least two other implementations despite of the workloads. We compared JEThread with pthread per core, pthread per task and Goroutines. Then, we test the scalability of JEThread by gradually increase the number of threads in the thread pool running $10,000$ short tasks. The result matches with our expectation as the scalability is achieved when increasing the number of threads in the thread pool.

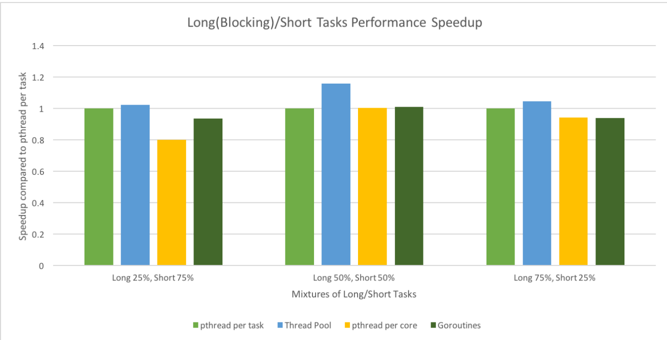

Fig. 1. Benchmark speedup using long blocking tasks and short tasks. From left to right each section represents: long 25% with short 75%, long 50% with short 50%, and long 75% with short 25%. Green bar: pthread per task. Blue: Thread pool. Yellow: pthread per core. Dark Green: Goroutines

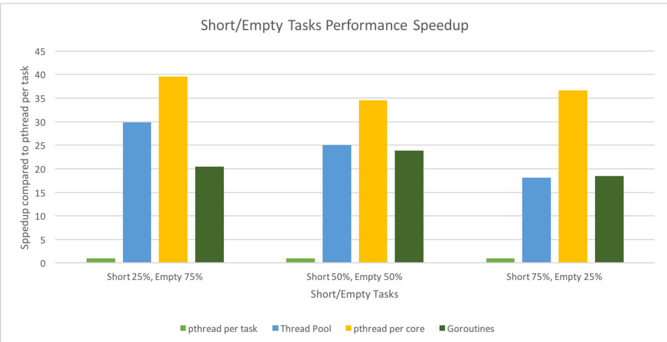

Fig. 2. Benchmark speedup using short blocking tasks and empty tasks. From left to right each section represents: long 25% with short 75%, long 50% with short 50%, and long 75% with short 25%. Green bar: pthread per task. Blue: Thread pool. Yellow: pthread per core. Dark Green: Goroutines

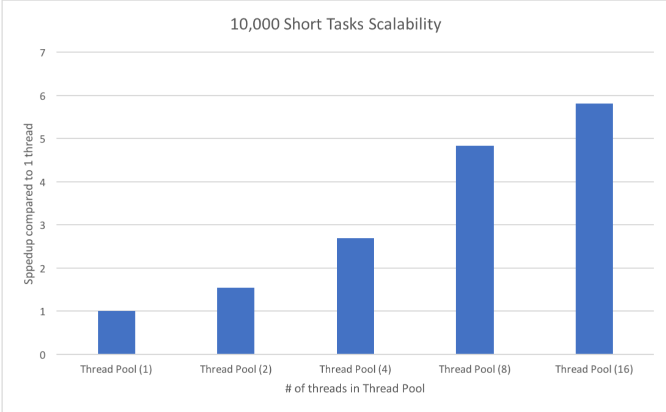

Fig. 3. Benchmark scalability of thread pool with $10,000$ tasks running. Threads used from left to right: 1, 2, 4, 8, 16.

VII. Conclusion

A thread pool should be efficient when managing a large number of tasks and executing them concurrently. In JEThread, we use work stealing to facilitate 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 transferred to each thread's task queue as executing tasks requires dequeueing the task each time. We've also conducted experiment with creating only one task queue shared between all threads, but the same thing still applies here that the overhead is transferred 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 unaware 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 and cause the main thread waiting unnecessarily. Additional optimization such as runtime dynamic workload check and compiler support for user-guided optimization will further increase the usability of thread pool.

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