15-618: Parallel Computer Architecture and Programming
Final Report: A Preemptive User-Level Thread Library

Michael Xu (Andrew ID: yijuex)
Yao Xu (Andrew ID: yaoxu)

May 2023

Project Website Link

1 Summary

A user-level thread library is implemented in this project. Users could use interfaces including uthread_create and uthread_join to create and rape user-level thread maintained by the library, which makes it easy for programmers who had experience in writing multi-thread parallel program with pthread to create programs with user level parallelism. Also, interfaces like umalloc, uprintf and uthread mutex lock are also provided for users to ensure user level thread safe. This thread library can be used for programming parallel programs on machines that run MacOS or Linux with X86 architecture CPU. We also did test on the correctness and scalability on the system.

2 Background

Concurrent programming is increasingly necessary due to the rise of multi-core processors, high-performance computing requirements, scalability needs, user expectations for responsiveness, and cost efficiency considerations. Multi-threading has become the go-to approach for concurrent programming due to its lower resource consumption and the exploitation of the shared memory model. While POSIX threads are widely used for multi-threading implementations, as they are efficient, scalable, reliable, and available on many operating systems, using them to achieve efficient concurrency remains a challenge.

One challenge arises when the number of concurrent tasks increases, potentially causing significant scheduling overhead and overwhelming hardware resources. The thread pool technique has long been used to address this issue, where it abstracts each concurrent task into a unit of work and adds it to a shared work queue. From this queue, threads in the pool grab one unit of work and finish executing it, then proceed to the next until the queue is empty.

The thread pool can be a neat solution to deal with a huge quantity of tasks while keeping the scheduling overhead bounded. However, it may be insufficient when the running spans of each concurrent task are highly variable. This is especially true if we aim to achieve user-level multitasking, where we can have long-running or persistent tasks. The thread pool may not be a wise solution here, as the tasks that run much longer than the others can cause drastic delay to those far down in the queue, which, in the worst case, may never be executed. That's because in the thread pool model subsequent tasks are not executed until all preceding tasks have been completed. If we were still to use pthreads naively, we would end up spawning hundreds of threads again, one for each task, producing immense overhead and hurting performance.

Clearly, the thread pool model should be preserved. But how to break the execution of long-running blocking tasks without compromising correctness to allow other tasks to run? To answer this question, we must introduce the idea of user-level threads.

In this project, we turn each unit of work in the work queue into an execution context. Each worker thread (POSIX thread) has a task queue stores user level tasks that assigned to it when user create a user level thread with uthread_create. These user level tasks are structures

where we turn each unit of work in the work queue (we are still sticking to the thread pool model) into an execution context, keeping track of the full state of each concurrently running task, and instead of mapping one task to one dedicated POSIX thread, we dynamically decide which POSIX thread in the pool should execute which task, so that we can safely break the execution of one task, saving its context back to the shared queue, and start the execution of the next user-level thread, loading the corresponding context from the queue.

3 Approach

Before going into the core part of this project, some initialization and helper function will be introduced first. This user level thread library requires some initialization before starting to create and schedule user level threads. The runtime_start function shown below is the first function will be called to initialize a structure called **runtime** which is defined in uthread.c file that contains some run time information. First, there will be only one thread enter this function. We record this main thread as the **master worker thread (pthread)** and mark the start flag to one so that the runtime will only be initialized once. Then, in the for loop below, as many worker threads (pthread) as there are CPU cores will be created.

From a perspective of the user level thread library, these worker threads are managed by **struct worker**. There are three important data structure associated with each worker thread which are **ID, pthread_id and a linked list contains struct uthread**. ID is a number from 0 to number of cores - 1 which is assigned when creating the worker thread. pthread_id is just a pthread_t type number which returned when pthread_create function is called. The linked list contains structures (**struct uthread**) that describes each user level thread assigned to this worker. How user level thread is defined and implemented in this library will covered later in much more details. (line 1-line 28)

After initializing the run time structure and all worker threads, we installed two signal handlers for SIGALRM

and SIGUSR1 for worker threads. SIGALRM combined with SIGUSR1 is acting as a timer which is used to notify worker threads to schedule a new user level thread from its linked list. (line 31-line 58)

```
1  void runtime_start() {
2      runtime.master = pthread_self();
3      runtime.started = 1;
4
5      // get core count
6      runtime.worker_count = (uint32_t)sysconf(_SC_NPROCESSORS_ONLN);
7
8      // start as many works as cores
9      // allocate memory for workers
10     runtime.workers = _umalloc(sizeof(struct worker) * runtime.worker_count);
11     // initialize workers
12     for (int id = 0 ; id < runtime.worker_count; id++) {
13         struct worker *w = &runtime.workers[id];
14         // install worker id
15         w->id = id;
16         // initialize the work queue
17         // create a dummy uthread that will stay as the head of the work queue
18         w->head = _umalloc(sizeof(struct uthread));
19         // initialize the dummy uthread
20         // this dummmny uthread does not have an id and will never be exposed to the user
21         w->head->next = w->head; // circular queue
22         memset(&w->head->ucon, 0 , sizeof(ucontext_t));
23         // set the dummy uthread as the current thread in execution
24         w->head->state = USTATE_RUNNING;
25         w->cur = w->head;
26         // start the pthread
27         pthread_create(&w->pthread_id, NULL, dummy, (void *)(uint64_t)id);
28     }
29
30     // install signal handlers
31     struct sigaction sa_alrm, sa_usr1;
32
33     memset(&sa_alrm, 0, sizeof(sa_alrm));
34     memset(&sa_usr1, 0, sizeof(sa_usr1));
35
36     sa_usr1.sa_flags = SA_SIGINFO | SA_RESTART;
37     sa_usr1.sa_sigaction = scheduler;
38     sigfillset(&sa_usr1.sa_mask);
39     sigaction(SIGUSR1, &sa_usr1, NULL);
40
41     sa_alrm.sa_flags = SA_RESTART;
42     sa_alrm.sa_handler = sigalrm_handler;
43     sigfillset(&sa_alrm.sa_mask);
44     sigaction(SIGALRM, &sa_alrm, NULL);
45
46     // create a periodic timer
47     struct itimerval timer;
48
49 #define MS (1000)
50     // Configure the timer to fire every 10ms
51     timer.it_value.tv_sec = 0;
52     timer.it_value.tv_usec = MS * 10;
53     timer.it_interval.tv_sec = 0;
54     timer.it_interval.tv_usec = MS * 10;
55
56     // start the timer
57     while (runtime.ready_count != runtime.worker_count);
58     setitimer(ITIMER_REAL, &timer, NULL);
```

```
59 }
```

Following function sigalrm_handler defined in uthread.c described how these two signals are used in terms of scheduling user level threads. SIGALRM is generated by the OS based on the scheduling round period we defined when setting the timer. Since the OS will send the SIGALRM to a random worker thread that is running in the current process context and we want all worker threads to know they should schedule the next user level thread, the thread receives SIGALRM will send a SIGALRM signal to the **master worker thread** and this master worker thread will send SIGUSR1 to each of other worker threads to notify them a new scheduling round comes.

```
1  void sigalrm_handler(int signum) {
2      if (pthread_self() != runtime.master) { // not master
3          pthread_kill(runtime.master, SIGALRM); // forward SIGALRM to the master thread
4      } else {
5          // bcast SIGUSR1 to all workers to start the scheduler
6          for (int i = 0; i < runtime.worker_count; i++) {
7              pthread_kill(runtime.workers[i].pthread_id, SIGUSR1);
8          }
9      }
10 }
```

The second signal handler which is used to process SIGUSR1 is registered as a function called **scheduler**. Back to **runtime_start** function, this signal handler is installed by sigaction with a flag called **SA_SIGINFO**. When the SA_SIGINFO flag is specified in act.sa_flags, the signal handler address is passed via the act.sa_sigaction field. This handler takes three arguments, as follows:

sig The number of the signal that caused invocation of the handler.

info A pointer to a siginfo_t, which is a structure containing further information about the signal, as described below.

ucontext This is a pointer to a ucontext_t structure, cast to void *. The structure pointed to by this field contains signal context information that was saved on the user-space stack by the kernel.

This is why our SIGUSR1 handler (scheduler) has three arguments and the third argument is very important because of the following reason:

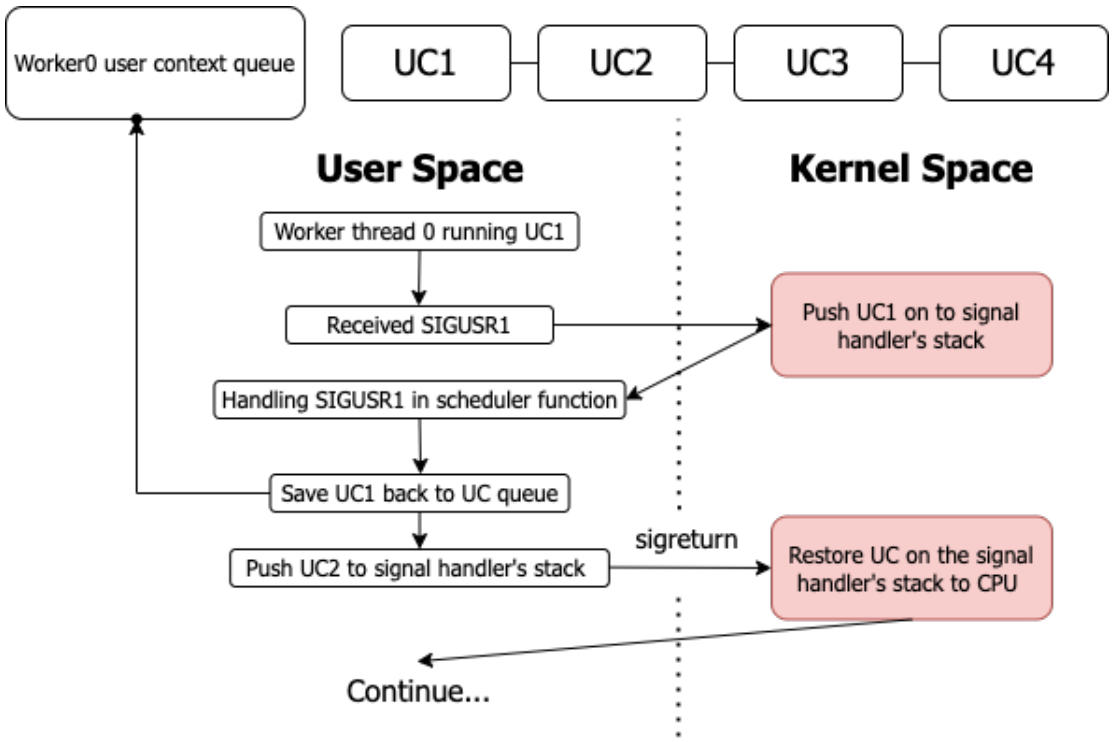

Figure 1: Process of user level scheduling

The above graph shows the core process of user level scheduling. First, "UC" stands for "user level thread context".

```
1  struct uthread {
2      int id;
3      int state;
4      char *stack;
5      void *retptr;
6      void *aux_rsp;
7      char *aux_stack;
8      ucontext_t ucon;
9      struct uthread *next;
10 };
```

User level thread context is maintained with a structure called struct uthread defined in uthread.c. Each user level thread has its own ID, state, stack, a ucontext_t struct and a pointer to the next struct uthread in the current queue. ucontext_t is a structure defined by POSIX used to save context including general purpose register and SIMD registers etc. Back to the above graph, when a worker thread received SIGUSR1, the operating system will push three inputs for the signal handler onto its stack which can be used within the signal handler. The third one is the new user level context that was running on CPU. It will be casted to a ucontext_t type variable and saved back to the user level thread context queue bu the scheduler. It will also push the next scheduled user level thread context to the stack to replace the user level thread context

pushed by the OS. When signal handler returns, a system call called sigreturn will store the new context back to CPU and continue execuation, and user level context switch is achieved.

Two interfaces most related to users are uthread_create and uthread_join. The following chunk of code is the implementation of the uthread_create function. For each newly created user level thread, a stack is allocated on the heap for later execution. The execution context will also be prepared by setting RSP to newly allocated stack, setting RDI to input and setting RIP to the function that user would run. On Linux, code segment register and stack segment register are also need to be set manually since Linux will restore context to CPU with value specified in the ucontext_t structure including CS and SS register.

On Linux, there is a mcontext_t structure defined inside of the ucontext_t structure under usr/include/x86_64-linux-gnu/sys/ucontext.h. This structure defines all registers that can be managed by using ucontext. User can access or change REG_CSGSFS to manage CS and SS register. However, there is a invisible padding inside of this field to user which took us a while to figure out how to store CS and SS register. Unlike Linux, MacOS manage CS and SS register automatically.

Finally, a round robin algorithm is used to assign new user context to work thread. A lock free operation is used to add newly created user level context after the first element in a worker's queue by using _cas function defined in asm.s.

```
1
2  #define UTHREAD_STACK_SIZE (1024 * 1024 * 8)
3  #define UTHREAD_SCRATCH_SPACE_SIZE (1024 * 4)
4  void uthread_create(uthread_t *id, void *(*func)(void *), void *arg) {
5
6      if (!runtime.started)
7          runtime_start();
8
9      struct uthread *u = _umalloc(sizeof(struct uthread));
10     memset(&u->ucon, 0, sizeof(ucontext_t));
11     // assign id
12     u->id = runtime.next_uid++;
13
14     // allcoate stack on heap
15     u->stack = _umalloc(UTHREAD_STACK_SIZE);
16     // initailze the stack
17     uint64_t *rsp = (uint64_t *)(ALIGN16(u->stack + UTHREAD_STACK_SIZE) - 8);
18     // prepare the return address
19     *rsp = (uint64_t)cleanup;
20
21     // allocate the aux stack
22     u->aux_stack = _umalloc(UTHREAD_SCRATCH_SPACE_SIZE);
23     // initailize the aux stack
24     u->aux_rsp = (void *)(ALIGN16(u->aux_stack + UTHREAD_SCRATCH_SPACE_SIZE)); // do not
           subtract 8 here
25
26     // initialize the execution context
27     u->ucon.uc_mcontext.gregs[REG_RSP] = (greg_t)rsp;
```

```
28      u->ucon.uc_mcontext.gregs[REG_RDI] = (greg_t)arg;
29      u->ucon.uc_mcontext.gregs[REG_RIP] = (greg_t)func;
30      u->ucon.uc_mcontext.gregs[REG_CSGSFS] = _cs() | ((uint64_t)_ss() << 48);
31
32      // initialize the state
33      u->state = USTATE_SLEEPING;
34
35      // pick a worker and insert the uthread into ist work queue
36      struct worker *w = &runtime.workers[runtime.next_worker++ % runtime.worker_count];
37      struct uthread *head = w->head;
38      // add to the work queue (always add after the head)
39      struct uthread *old_next;
40      do {
41          old_next = head->next; // the next uthread of the head is guaranteed to be valid
42          u->next = old_next;
43      } while (_cas(&head->next, (uint64_t)old_next, (uint64_t)u) != (uint64_t)old_next);
44
45      *id = (uthread_t)u;
46  }
```

There are three function in this library are about resource recycling. **cleanup** function is the return address after of each user level thread, it set the current user level thread to USTATE_JOINABLE. User level thread with USTATE_JOINABLE won't be recycled immediately. After uthread_join is called, user level thread will be set to USTATE_JOINED.

In each worker's queue, the dummy head is a user level context that is responsible for freeing resources. When this user level context is scheduled, it will look for user level context that is in USTATE_JOINABLE and USTATE_JOINED. If the uthread is in USTATE_JOINABLE, the stack will freed because the JOINABLE uthread cannot possibly be running since they are mapped to a single worker who's only able to do one thing at a time. That means the uthread will no longer be schedulable from now on because it can only be run if the scheduler was to schedule it later, which will not happen, whereby it is safe to free its stack and execution context. If the uthread is in USTATE_JOINED, the struct uthread will be freed since when dummy is running, the scheduler must not be running since they again share the same worker, thus, if dummy sees one uthread in the JOINED state, that uthread cannot be scheduled to run again, since first the scheduler is uninterruptible, which eliminates the intermediate state being observed by dummy and second the next time scheduler tries to schedule it will definitely also see it as JOINED and do nothing. However, one thing to notice is that we have to make the JOINED uthread unreachable and then deallocate the memory otherwise, when the scheduler step through it may encounter invalid memory.

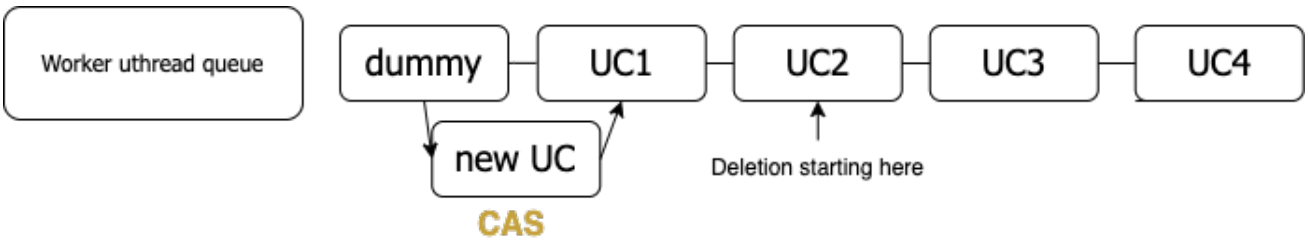

Figure 2: Ucontext adding and deleting from queue

The following graph shows the life cycle of a user level thread. In summary, a stack and other necessary data structures are allocated on the heap first. Then, the ucontext structure will be added into the queue of one of the workers and waiting to be scheduled. After execuation finished, it will jump to clean up function and wait to be joined and resource recycled.

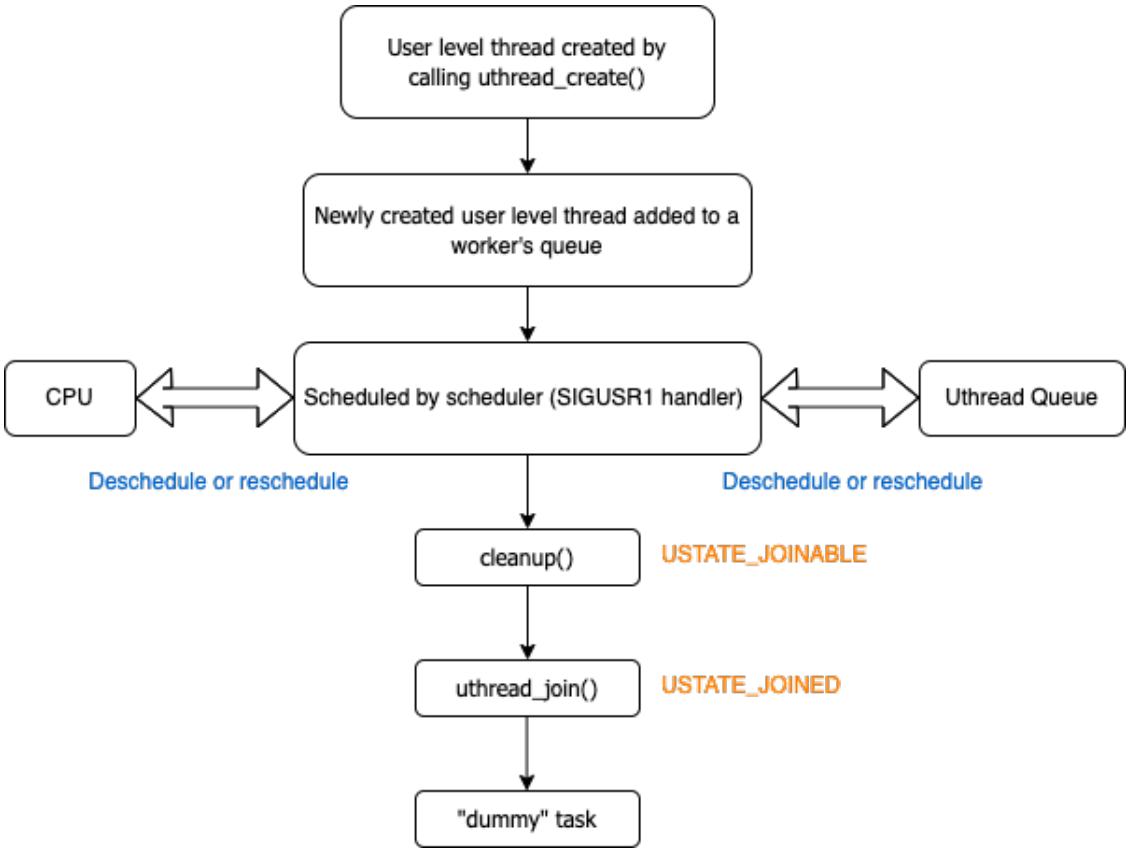

Figure 3: Lifecycle of a uthread

Here I wish to mention a few obstacles we have met till now. Initially, we thought of implementing work queues using doubly-linked lock-free lists; however, it turned out that there had been yet no known solutions

to this problem, which would not have been if a compare-and-swap instruction had been given by the architecture that can operate on two discounted addresses, and most of the articles and papers being widely accepted were just singly-linked lists. Thus, we decided to not go any further but to fall back to singly-linked lists, which did not appear to be any simpler, especially when it came to the deleting part, investigating which we got to know about the differences between lock-free and wait-free. After that, we decided to get around that, avoiding a deletion from happening in parallel with insertions or another deletions. That was achieved by constraining the insertion to only taking place between the head of the list and the second to the head. As for the deletion, we spawn another user thread during the run-time initialization that's not exposed to the user and is solely responsible for performing the deletion, and we call it the garbage collector, and an important constraint we put on this to prevent deletions and insertions from interfering each other on the second node, we forced the deletion to always start from the third node. In this way, we have a lock-free singly-linked list that can be added to in a concurrent lock-free manner. Then, when implementing uthread_join, and uthread_detach that involed traversing the work queue to verify the validity of a given uthread ID, we considered using the Linux RCU to better read-mostly efficiency and implemented it, but we eventually decided to revert back to having the user responsible for properly managing the IDs since the help of the RCU in our case was not conspicuously seen while it largely added to the code complexity and made the code potentially more error-prone in later stages. Then we faced the serious issue of not freeing the memory fast enough to make room for newly spawned threads, to which we came up with the solution that has each uthread free its own stack during the clean-up phase (a snippet of inline Assembly freeing the stack and saving the return value without needing a stack via using munmap syscall exploiting the fact that the syscall happens on the kernel stack).

4 Results

First test (following two graphs) is using fixed amount of uthread with different number worker thread to check the scalability of this library on machines with different of number of cores. We created 4000 uthreads that each calculates Fib 30 with different number of workers. This test is done on GHC machines(CPU: i7-9700 8 cores). Figure 4 shows the total execution time of Fib 30 with 4000 uthreads and 1,2,3,4,5,6 and 7 workers. Figure 5 shows the linear speed up in term of doing same amount of work when we are using more workers, which meet our expectation since more workers means more execution unit available for uthreads and more concurrency, just like more cores available for pthreads. It shows the correctness of user level

context switching, efficiency of the scheduling mechanism and good scalability of this library on a 8 core machine.

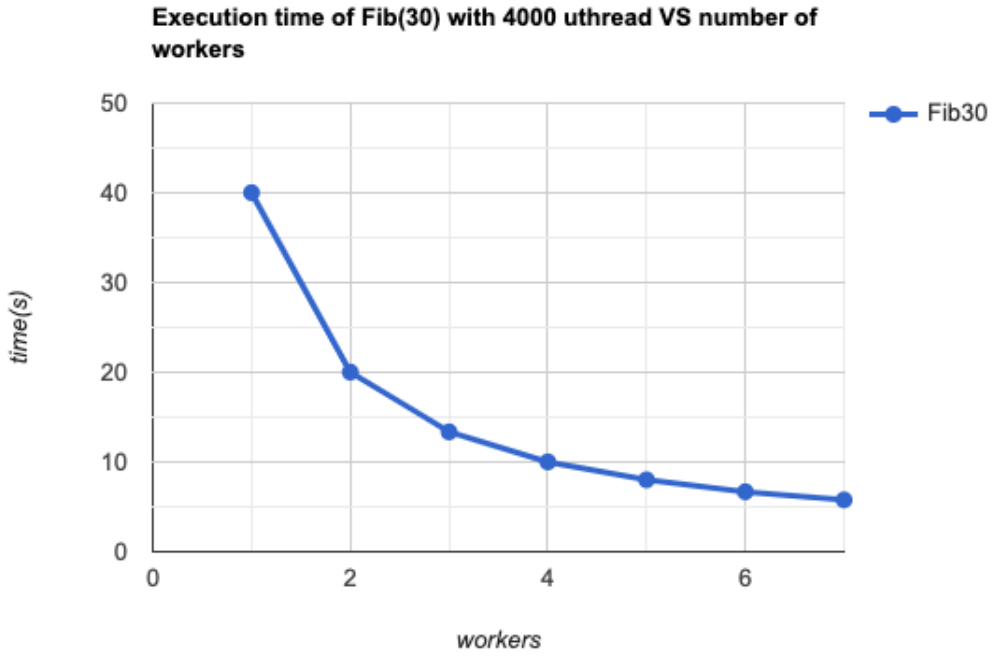

Figure 4: Execution time of Fib(30) with 4000 uthread

Figure 5: Speedup of execution time of Fib(30) with 4000 uthread

The second test is using fixed amount of workers with different number of uthreads to check the scalability of this library. We created 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000 uthreads that each calculates Fib 30 based on same number of workers. This test is done on GHC machines. Figure 6 shows the execution time of Fib 30 with 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000 uthreads and 7 workers. We can see the execution time is increasing in a linear way. This means when users are creating more uthreads, the system can provide stable scheduling and performance without meeting any memory or scheduling bottleneck. In extreme case, a system with same hardware configuration as GHC machines can support 200000 uthreads running at the same time.

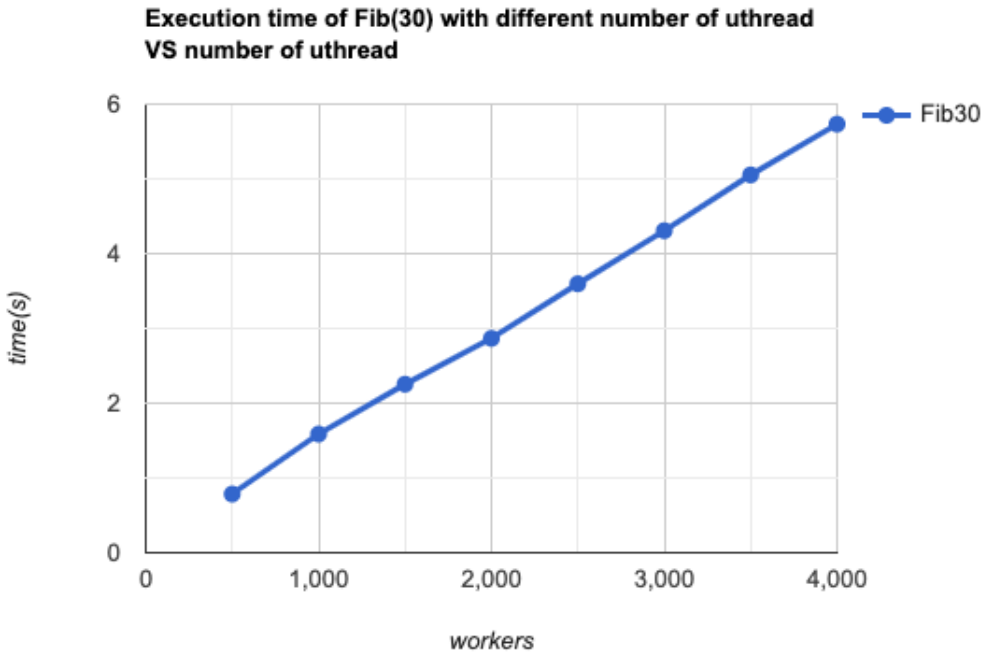

Figure 6: Execution time of Fib(30) with 4000 uthread

5 Reference

Sundell, H., & Tsigas, P. (2008). Lock-free deques and doubly linked lists. Journal of Parallel and Distributed Computing, 68(7), 1008–1020. https://doi.org/10.1016/j.jpdc.2008.03.001

6 Work distribution

Work is divided 50% to 50% in this project.