Work stealing Fork Join Framework in C++

Xin Jin and Jiarui Li

Carnegie Mellon University

December 10, 2021

Abstract

We implemented a C++ parallel framework for fork-join models in which problems are solved by divide-and-conquer algorithms. Those tasks that require splitting themselves into subtasks, waiting for them to complete, and then collecting results can easily take advantage of our framework to achieve approximate linear speedup. Our framework also allows users to choose from different stealing policies including continuation-stealing and child-stealing. Based on dynamic scheduling, our model benefits from workload balancing in terms of the performance under multiple threads. The measured performance plots show good parallel speedups for the problems we tested on, and also suggest possible improvements and future work that we can work on.

Keywords

parallel programming, fork-join model, work-stealing

1 Background

1.1 Divide and Conquer

Divide and conquer is one of the most straightforward problem solving techniques. This programming style recursively spawns subtasks, solves each of them in a separate activation record and finally composes the results. As the computation work of subtask becomes more and more expensive, at each subroutine calling point, it is reasonable to treat the subroutine and the continuation as two parallelable tasks and map each of them to parallel instances like threads to accelerate a broad range of problems. Because of the generality of this model, this parallel technique is called fork join paradigm.

Depending on the next executed task when calling fork, two kinds of implementation policies exist, i.e. the child first policy and the continuation first policy. The child first policy requires the threads to suspend the current execution, save the context of the current task, potentially give it to another idle thread (or store it in a queue) and switch to execute the subtask.On the other hand, the continuation first policy just suspends the spawned subtask and keeps executing the current task without storing the context. Furthermore, since fork join paradigm spawning subtasks in the runtime, a kind of dynamic scheduling is always applied. To be specific, in order to efficiently distribute tasks, the common practice is to use threadpool and distributed work queues, namely each thread will be associated with a dedicated task queue storing the tasks. In addition, when the work queue becomes empty, the corresponding threads can also steal from other threads' queues. Some famous implementations include Cilk[3],

FJTask[4], Hood[5].

In this project, we implemented both the child first and the continuation first policy. Our implementation differs from the above ones in that instead of modifying the compiler to manually save context and maintaining a global stack view using complex cactus stack structure and closure tree[], our model is much simpler to understand and implement.

Our continuation stealing model is based on the idea of the generator. To be specific, each task serves as a subtask generator and each time executing it will return a new subtask (if possible). Then it will suspend itself and wait to be called next time. We use stackless coroutines supported by modern C++ to implement it, which only requires about 300 LoCs. We also designed a new two queue technique to solve the circular dependency problem for the child first policy.

With sufficient parallable tasks and minimum contention, we expect our framework can achieve linear speedup relative to the number of parallel instances.

1.1.1 Stackful Coroutine & Stackless Coroutine

There are basically two kinds of coroutine implementation approaches, stackful coroutine and stackless coroutine.

For stackful coroutines, each coroutine requires its own stack and therefore can be suspended at any function depth level. However, it is quite hard to anticipate the to-be-used stack size and it is even harder to (potentially migrate the whole coroutine stack) migrate the coroutines across the threads. Even worse, since each coroutine needs a dedicated stack to maintain its own activation records, it limits the number of coroutines each thread can create. Currently, few implementations realize the cross-thread version, one of the examples is the Golang scheduler. Other implementations regard coroutines as a concept one level below the thread and do not allow the cross-threads migration behavior, therefore easier to implement, ex. boost.coroutine, boost.Fiber.

Another kind of coroutine is stackless, which means the coroutine stores its state in the heap and there is no need to maintain a stack for each coroutine when it is suspended. This approach depends on the fact that the size of the coroutine state can be determined in the compile time if its stack depth is only one. Apparently, the limitation of this approach is that only the top-level routine may be suspended to avoid stack increasing. But this approach requires less memory for each coroutine and has better scalability.

Fortunately, the fork join style typically follows the pattern that within each task, always (1) splitting the current task, (2) forking new subtasks, (3) joining them and (4) composing the results. Therefore, normally there is little need for non-top-level fork support. One step above, considering the implementation complexity and the scalability of our framework, we exploit stackless coroutine provided by modern C++ and a higher level coroutine primitive library called cppcoro, especially its generator primitive (expected to be included by the C++ standard library in the future).

1.1.2 Data Structures

The key design objective for our framework is to efficiently assign each task to execution resources. Our implementation is a lightweight, object-oriented fork join framework with modern C++ language level support.

From a high level, our core implementation is a work-stealing based distributed work queue model. To be specific, the key data structures include three parts. (1) a worker thread pool (2) a set of thread-dedicated work queues (3) the task base class. These data structures

are discussed separately in the following paragraphs.

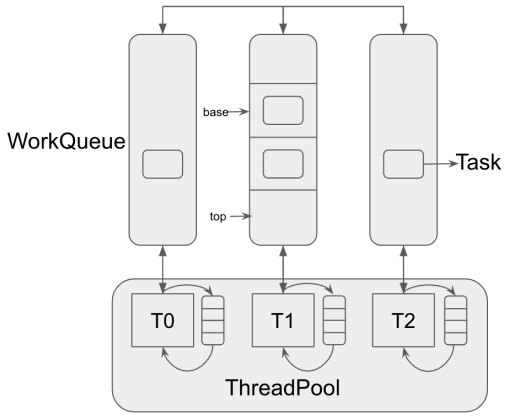

Figure 1: Core Data Structure

ForkJoin::ThreadPool To start a fork join task, users need to first instantiate a thread pool, whose thread size can be specified when constructed. It serves as a handle to a set of parallel execution resources. Under the hood, they consist of a set of pthreads. To start a fork join task, users should create a root task and pass to it. Besides, the threadpool class provides the forever loop executed by each worker thread and also the task stealing discipline.

ForkJoin::Task Task serves as the callable base class for users to derive. It provides the fork function for users to spawn parallely executed subtasks and join function for users to wait for their results.

ForkJoin::WorkQueue The WorkQueue class holds all the spawned tasks and each thread has its dedicated work queue. We implemented both a mutex lock guard version and a lock free version of it. Since the owner (owner thread) and the stealer (another idle thread) of the queue only access the queue from one side of the queue (like a deque), the class offers push() and pop() for the owner thread and task() for the stealer thread.

As a standard user case example, here is the Fibonacci function implementation using our framework.

```
class Fib: public Task {
private:
    int n;
    int result;
    static const int threshold = 20;
public:
    Fib(int _n): n(_n) {}
    int getResult() { return number; }
private:
    void operator()() override {
        if (number <= threshold) {
            number = seqFib(number);
            return;
        }

        Fib *f1 = new Fib(n-1);
        Fib *f2 = new Fib(n-2);

        fork(f1);
        fork(f2);
        join(f1);
        join(f2);

        result = f1->getResult() + f2->
            getResult();

        delete f1;
        delete f2;
     }

public:
    static int seqFib(int n) {
        if (n == 0) return 0;
        if (n == 1) return 1;
        return seqFib(n - 1) + seqFib(n - 2);
    }
};

int main()
{
    ThreadPool tp(4);
    Fib *root_task = new Fib(40);

    tp.startProcess(root_task);
    tp.destroy();

    cout << "result: " << root_task->getResult
        () << endl;
```

```
    delete task;
}
```

By deriving the base Task class, users pass the input through the constructor function. The intermediate execution can be stored in the instance variable and therefore accessible by the parent task after joining.

User is mainly able to tune two parameters similar to the [4] paper. First is the number of threads in the thread pool, whose default value is set to the maximum parallel ability of the machine using 'std::thread::hardware_concurrency()'. Second is the threshold granularity of the subtask for sequential execution, which can significantly affect the proportion of time for workers to keep stealing instead of doing useful work, i.e. the work efficiency of the threads.

2 Approach

2.1 Lock-free Work Queue

Our model has features of two different stealing policies, i.e. continuation stealing and child stealing, but our overall implementation for these two policies differs much. To solve the occurred issues and optimize them, we took different approaches.

The main challenges in designing work queues in child-stealing surround synchronization and its avoidance. Initially, when we were developing the child-stealing policy, we used fine-grained locks to synchronize distributed work queues and avoid any other concurrency issues. However, when we did any operations on the work queues, we always needed to acquire a lock which caused numerous contentions and degrade performance.

Obtaining locks for every queue operation can easily become a bottleneck. Therefore, instead of using fine-grained locks, we attempted to implement a lock-free work queue for regular queue operations like push and pop and stealing from other threads. If we only have a single reader and a single writer, it will be easy and straightforward to implement a lock-free deque as taught in class. However, in our case, a work queue can have two writers at the same time. One such scenario is that a worker thread is trying to push new work onto its own work queue, while another worker thread is stealing work from the same work queue.

At our first thought, we believed a deque was a good data structure for our work queue. To be specific, we could allow stealing from other threads to happen at one end of the work queue, to say the bottom of the queue which similar to the approach used in the [4] paper. Similarly, a worker thread pushes and pops tasks from the top of its own work queue. In a nutshell, stealing only happens at the bottom of a work queue, and push/pop happens at the top of a work queue. Notably, when a work queue only contains one task, the pop operation has a higher priority than stealing. In other words, in our implementation, a worker can never steal tasks from a work queue with only one single task, which makes our design of work queues simple. Furthermore, we declared the bottom and top of a work queue as atomic variables. So, we can check whether a work queue has more than one element by pre-decrementing top:
If ($--top >= base$) in pop operation
and pre-incrementing base:
If ($++base < top$) in steal operation.

Noticing that pop and steal operations can still interfere if the deque is about to become empty, we still need entry locks to grant the invoking thread access to the work queue for stealing. Likewise, a work queue can be 'fake' empty because we pre-increment the base of a work queue. So, we should do a double-check in pop. Especially, when a pop operation does the second check, it should acquire the entry lock for the work

queue to ensure that all stealing operations on that work queue have finished so far. This allows the worker thread to know if its work queue is truly empty. Besides the above scenario, pop and steal are guaranteed to operate on disjoint elements of the array without causing any synchronization issues.

```
void
WorkQueue::push(TaskInnerWrapper* task) {
    int t = top;
    top++;
    wkq[t] = task;
}

TaskInnerWrapper*
WorkQueue::pop() {
    int t = --top;
    if (t >= base){
        return wkq[t];
    } else { /* empty */
        entrylk.lock();
        if(base > t){
            /* true empty */
            ++top;
            entrylk.unlock();
            return NULL;
        } else {
            entrylk.unlock();
            return wkq[t];
        }
    }
}

TaskInnerWrapper*
WorkQueue::take(){
    entrylk.lock();
    int b = base++;
    if (base < top){
        entrylk.unlock();
        return wkq[b];
    } else {
        base--;
        entrylk.unlock();
        return NULL;
    }
}
```

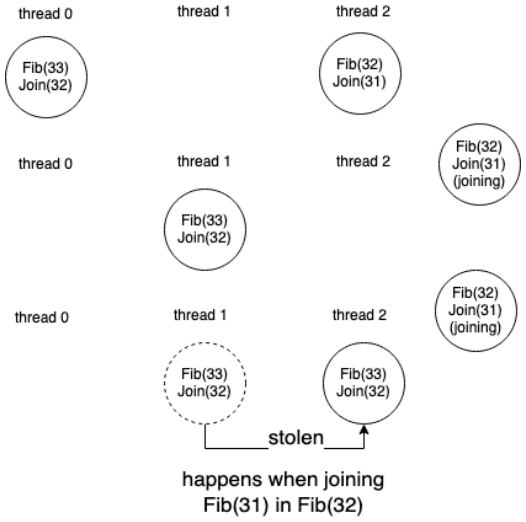

Figure 2: "Dead Joins" Illustration

2.2 2Q for Continuation Stealing

Regarding concurrency issues in continuation stealing, things get more complicated than child stealing, and the main challenge here is to avoid "dead-join"s. Since the stolen unit is not a task or subtask in continuation stealing, there exist dependencies among continuations. Additionally, our join method is greedy. When the task is waiting for its subtasks to finish, it tries to finish all the work from in its own work queue. If the work queue is empty, it will try to steal work from others. If it steals a task that has dependencies with the current task, the current join can be blocked forever.

To illustrate the 'dead-joins' problem, let's take the Fibonacci sequence program with n = 33 and three threads as an example which is shown in the Figure 2 . For simplicity, other tasks which are not related are not shown in the Figure 2. In the beginning, task Fib(33) was in the work queue of thread 0. When thread 2 began task Fib(32) which is at the step of joining Fib(31), thread 1 stole task Fib(33) from thread 0 and pushed it onto its own work queue. However, thread 2 found that Fib(31) was not finished then, and it tries to work on tasks in its own work queue. Finding that its own work queue was empty, thread 2 stole task Fib(33) from

thread 1. The severe issue occurred when thread 2 attempted to execute task Fib(33). Although thread 2 started to join Fib(32) in task Fib(33), Fib(32) could never be accomplished. The original context of thread 2 was joining Fib(31) in task Fib(32), but now thread 2 is waiting for Fib(32) in task Fib(33). Until task Fib(33) is finished, we can continue task Fib(32). But until task Fib(32) is joined, task Fib(33) can be accomplished. In other words, these two tasks are waiting for each other. The reason why this issue happens is that continuations can have dependencies. If continuations are stolen randomly, we can not guarantee that order of accomplishment is preserved in work queues.

This was the biggest challenge we had to overcome in this project. One approach to circumvent this issue is that we only allow threads to steal from one single queue. In other words, we could change our design for continuation stealing from distributed work queue to a single central work queue. At the start of the process, we push the root task onto that work queue, and every worker thread steals tasks from it. Since subtasks are generated in order in the central work queue, we don't have to worry about the order of continuation in this case. However, a central work queue is not a very efficient way to achieve dynamic scheduling. Since we still had entry locks for stealing as we mentioned before, a single work queue could easily be a bottleneck.

Another approach with distributed work queues is to have a local work queue for each thread, and we call this method "2Q". One thing to point out is that dependencies among different continuation only matter at the join but instead of a fork. In addition, we would like to do works when we are waiting for other tasks to be completed. Thus, instead of continuing executing tasks in the shared work queue or stealing tasks from other work queues in the context of joining subtask, we can simply push the current continuation (join(subtask)) onto a local work queue that nobody else can access if the waited subtask is not finished at that moment. We can visit this continuation later, but for now, the thread will simply suspend it and attempt to steal tasks from others. Significantly, we protect these continuations by pushing them in a local work queue. Therefore, we have two work queues for a thread. Namely, one is a shared work queue, and the other is a private work queue.

In a word, if a worker thread finds that the continuation is not finishable at that moment, it just pushes it to the back of its local work queue and tries to fetch the next task from its shared work queue. When the shared work queue is truly empty, it will try to fetch a task from its local work queue. If that one is still not finishable, the thread simply pushes back the continuation at the front of the local work queue and performs stealing.

```
while((curr_task = curr_workq.pop()) != NULL){
    // bookkeep befor executing task
    curr_task->task->tp = tp;
    curr_task->task->threadId = threadId;

    if(curr_task->gen_it == NULL){
        /* task has not been started */
        curr_task->gen_it = make_unique<
            cppcoro::detail::
            generator_iterator<Task *>>(
            curr_task->gen->begin());
    } else {
        /* task is conitunuation */
        (*curr_task->gen_it)++;
    }

    if ((*curr_task->gen_it) == curr_task->gen
        ->end()){
        curr_task->task->is_done = true;
        Task *temp_task = curr_task->task;
        delete curr_task;
        if ( temp_task == task ) return;
    } else {
        curr_workq.push(curr_task);
```

```
        Task *child_task = *(*curr_task->
            gen_it);
        TaskInnerWrapper *child_task_wrapper =
             new TaskInnerWrapper{
            make_unique<cppcoro::generator<
                Task *>>((*child_task)()),
            NULL,
            child_task
        };
        threadId, child_task_wrapper->task->
            getId());
        curr_workq.push(child_task_wrapper);
    }
    tp->steal(threadId);
}
```

3 Results and Analysis

We used the computing resources in the Pittsburgh Supercomputing Cluster (PSC) to evalute our implementation. To be specific, we use the non-shared nodes in the cluster, which are equipped with two AMD EPYC 7742 processors (2.25-3.40 GHz, 2x64 cores), 256 GB RAM and 256MB L3 cache.

We mainly used our framework to accelerate three typical applications. First is to recursively calculate the 50th term of the Fibonacci sequence, with the sequential calculation threshold equals to 30. Second is matrix multiplication. For simplicity, we tested our model with 2 2048x2048 identity matrix with granularity of 64x64. Third is to use quick sort to order 1,000,000,000 integers with the sequential calculation threshold equals to 1,000,000. We ran both the child-stealing policy and the our 2Q implementation on these applications in the cluster.

3.1 Speedup

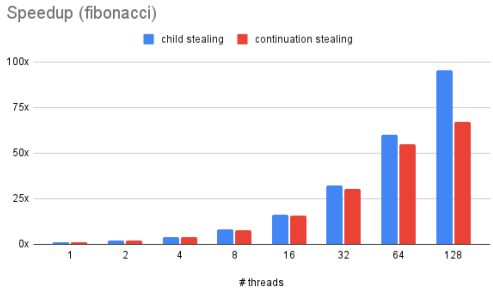

Figure 3: "Speedup in Fibonacci"

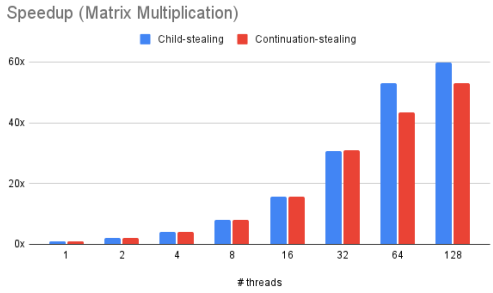

Figure 4: "Speedup in Matrix Multiplication"

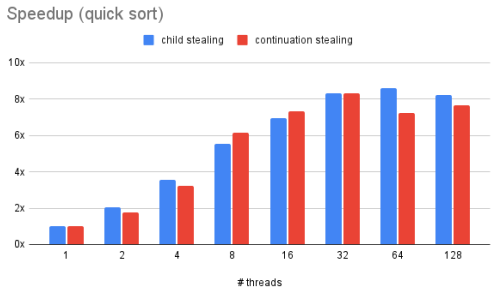

Figure 5: "Speedup in Quick Sort"

3.1.1 Task Granularity

In the problem of the Fibonacci sequence, we started with a small task granularity. That is when a task is under a user-defined threshold, that task can no longer be divided, and we work on that task directly instead of dividing it into smaller pieces. In this case, the threshold of task granularity is the input number. However, we found the performance was not good as

we expected since we distributed more work to worker threads. Eventually, we increased the granularity to achieve good performance on this task. The benchmark started with 50, and the threshold was 30. The program was sped up linearly with the number of threads growing up before the number of threads got to 64 for both two stealing policies. Although we allocated more threads in our thread pool, the program was not accelerated to an ideal speed, especially for the continuation-stealing. Though the speed actually increased, it still struggled when we increased the number of threads from 64 to 128. In this case, child-stealing performed better compared to continuation-stealing.

3.1.2 Stealing Granularity

On one hand, we believe that we may not get a great threshold value for this problem. Because in our previous testing, the threshold value actually mattered the overall speed. As we mentioned earlier, a bad threshold can lead to bad speed which could be worse than the sequential program. On other hand, more contention in continuation-stealing can be one factor. In child stealing, the stealing granularity is always a task. However, in continuation-stealing, the stealing granularity is a continuation that has less workload than a single task for most of the time. Since, in our design, every work queue except the thread 0's work queue is empty at the start of the process. So, stealing can happen at an early stage of the whole process. A small stealing granularity may lead to more contention potentially in our design.

3.1.3 Trade-off Between Task Granularity and Contention

Task granularity is an important concept in our design with the divide-and-conquer algorithm because it can affect greatly the contention. From our point of view, the reason why a small task granularity resulted in bad performance is that numerous contentions were dragging down the speed. Considering a scenario that the task granularity is very small, this actually is good for workload balancing since the workload is distributed in a find-grained way. However, the work queue of workers can be empty very soon, since they finish their work faster compared with a large task granularity. In other words, stealing happens more frequently. Although we have semi-lockfree work queues for workers, we still have entry locks for stealing. Numerous contentions for acquiring entry locks can be a great cost in this case. However, considering task granularity is a little bit out of our scope, since task granularity is complicated and different for different applications, and we only provide a framework for users.

3.2 Stealing Overhead

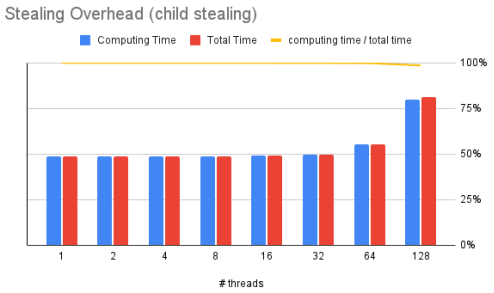

Figure 6: "Stealing Overhead in Child Stealing"

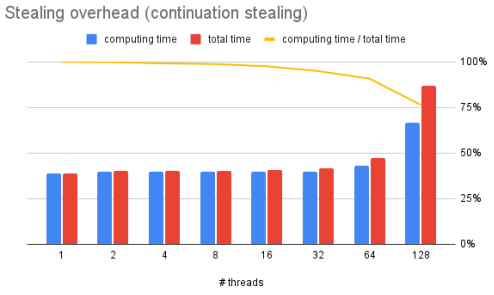

Figure 7: "Stealing Overhead in Continuation Stealing"

Stealing plays an important role in our model, so we would like to see the influence of thread numbers on the stealing overhead. In order to compare the overhead in stealing with the different number of threads, we would like to compare the overall execution time and the overall processing time to represent total work and total task size. To be more specific, we sum all the execution time and computing time across threads. Since we did not change the overall task size and the number of tasks, these two values should be consistent when we changed the thread number. Notably, the result matched our expectations before we had more than 64 threads. Conversely, the overall time rose dramatically when the thread number reached 128, especially for continuation stealing. To get a deeper insight, we also plotted the ratio of execution time to the processing time. It turned out the threads did less effective work under a large thread number, and the total task size increased.

In our consideration, having more threads means fewer probabilities to steal work successfully in the early stage of the process. Like we mentioned before, most work queues are empty at the beginning of the process, and workers steal randomly. Therefore, there will be less chance for an idle worker to choose a non-empty work queue as a victim in our design. This fact potentially increases the total number of stealing significantly which causes large stealing overhead as we can observe in the figure 6 and figure 7. By the same token, when the process is about to finish, most work queues are empty, it is also hard for a worker to steal work. However, we didn't have further evidence to prove this affect the overhead greatly, and we may need to explore it further in our future work.

3.3 Work Efficiency

In our fork join framework, the running time of each thread includes two part, stealing time and computing time. Therefore, the key metrics for work efficiency is each thread's computing time percentage relative to the overall running time. Followings are several measurements of the ability of our framework to balance the workload when there are eight threads.

Figure 8 through 11 shows near optimal balanced load for applications like Fibonacci calculation and matrix multiplication. All threads' computing time equal to their running time. This result also matches the linear speedup result in section 3.1. Since each thread devote their full time to do useful work, the speedup should be 8x which means about 100% work efficiency.

For quick sort, results in figure 12 and figure 13 shows that each thread's work is not so even. With thread 0's computing time tends to take over larger percentage than others. We believe it is because it starts the root task and spawns the first several subtasks. On the other hand, other threads' work efficiency tend to be similar and it may be due to the fact that each thread randomly steal tasks from other threads' work queue, therefore with similar probability to successfully steal a task, i.e. similar stealing time. As for the difference between child stealing and continuation stealing policy, our two queue implementation of the continuation stealing policy tends to have less work efficiency. We believe it is because we store those waiting to be joined tasks in local queue instead of letting other threads to check whether their waiting tasks finished or not, which introduces more latency.

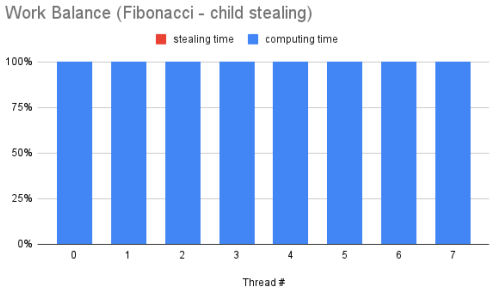

Figure 8: "Work Balance (Fibonacci with child-stealing)"

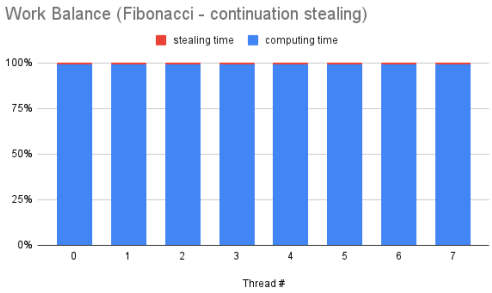

Figure 9: "Work Balance (Fibonacci with continuation-stealing)"

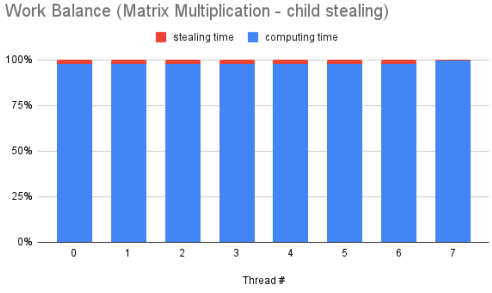

Figure 10: "Work Balance (Matrix multiplication with child-stealing"

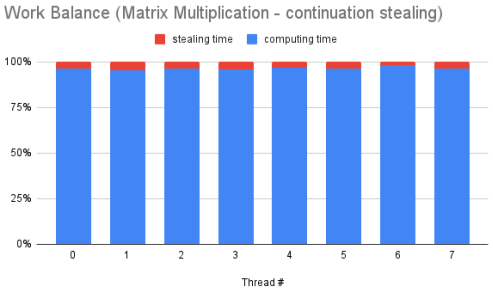

Figure 11: "Work Balance (Matrix multiplication with continuation-stealing"

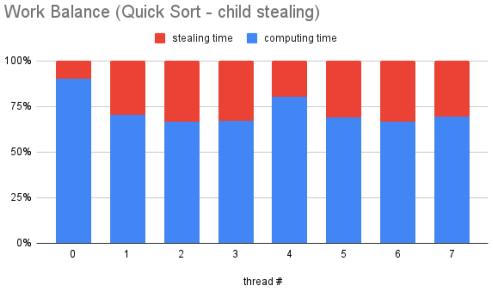

Figure 12: "Work Balance (Quick Sort with child-stealing"

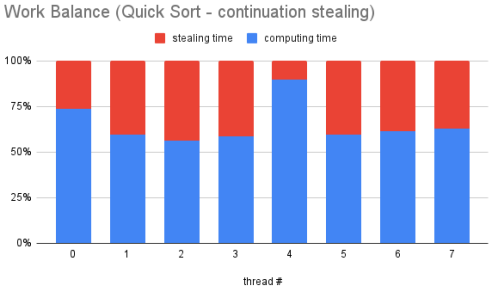

Figure 13: "Work Balance (Quick Sort with continuation stealing)"

4 Future Work

Because of the limit of time, we haven't compare the performance of our framework with other implementations, like cilk[3], Hood[5], JTask[4] etc. And it is meaningful to compare our coroutine-based implementation with the compiler-modification based implementation of cilk.

In addition, we are still assessing the necessity to extend our framework to a distributed version, which can fork task and steal task from another process and probably in another machine. This version will have much more difficulties, since we should further consider the state migration cross machines, the communication mechanism and even the failure recovery mechanism.

4.1 Work Distribution

The project was distributed 50%-50%. We worked together on the implementation of child-stealing and continuation-stealing. However, we measured the performance of our model separately. Xin focused on matrix multiplication, and Jiarui worked on quick sort.

References

[1] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 356-368, doi: 10.1109/SFCS.1994.365680.

[2] Färnstrand, Linus. "Parallelization in Rust with fork-join and friends: Creating the fork-join framework." (2015).

[3] Frigo, Matteo Leiserson, Charles Randall, Keith. (1999). The Implementation of the Cilk-5 Multithreaded Language. ACM SIGPLAN Notices. 33. 10.1145/277650.277725.

[4] Lea, Doug. (2000). A Java Fork/Join Framework. ACM 2000 Java Grande Conference. 10.1145/337449.337465

[5] Blumofe, R. and D. Papadopoulos. "Hood: A user-level threads library for multiprogrammed multiprocessors." (1998).