Carnegie Mellon 15-418 Spring 2024 Final Project Milestone Report: Cilk Fork-Join Parallelism Library

Yonah Goldberg
ygoldber@andrew.cmu.edu

Jack Ellinger
jellinge@andrew.cmu.edu

Project Web Page

Work Completed

We implemented two simple thread schedulers. Each scheduler ascribes to the following API:

- `void init(int n) // initialize the scheduler with a size n thread .pool`
- `std::future<T> spawn(std::function<T()> func) // spawn a function to be potentially run in parallel. Returns a future on the returned result.`
- `T steal(std::future<T> fut) // steal work while waiting for fut to finish. Returns the value of fut.`
- `void cleanup() // cleanup thread pool and join threads`

We might tweak this API as we make new schedulers, but for now, this works well. All tests run a generic scheduler, so we can easily swap out schedulers when benchmarking. Currently, we have tests for fib, quicksort, nqueens, and rectangular matrix multiplication. We run tests via the Google benchmark library.

The first scheduler we made for our Cilk library is called the simple scheduler. It spawns a thread to run the function every time the user calls `spawn`. `steal` waits on the future to finish and returns the result.

The next scheduler we made is called the simple child stealing scheduler. In this implementation, we have a thread pool where each thread has a work queue. The threads will attempt to take work from their queue, and if they have no work to do, they will try and steal work from other queues. We synchronize via mutexes. When the user calls `spawn`, we push the function to that thread's work queue. When the user calls `steal`, we steal functions from work queues to run until the future is ready.

After benchmarking this implementation, we began work on both a lock-free implementation of the child stealing scheduler and a continuation stealing scheduler. These implementations are still incomplete, but we have made great progress on both. For lock-free, we have code that looks like it is doing the compare-exchange properly, but we still need to fix some bugs. For continuation stealing, we found a way to use `setjmp` and `longjmp` to properly save and resume function execution. When we run a function, we first allocate space on the heap and point the stack pointer to the heap. If we want to save our execution context, we can call `setjmp` and we know the current function stack will not be corrupted since it is on the heap. Later, another thread can "steal" the continuation and resume the function. We have continuation stealing working on a single threaded implementation, and are now looking to make it work across multiple threads.

Deliverable Progress

Currently, we are right on track with our goals as stated in the proposal. At this point we wanted to have our thread pool/child-stealing and simple scheduler completed, which we have done. This week's

goal in our proposal is to try and implement contuinuation stealing, which we have been working on all weekend, and we have made great progress on one of our nice-to-haves (the lock free implementation).

Poster Session Plan

The poster is still planned to showcase speedup graph comparisons between our different implementations and OpenCilk across an array of problems. We can also show code snippets demonstrating how easy it is to parallelize code using our API, which is the main benefit of fork-join parallelism.

Preliminary Results

The following table shows preliminary results for our two working schedulers on a variety of test cases. Each benchmark shows the scheduler run, which problem it ran, and the problem size. The google benchmarking library automatically runs many iterations when it takes little time to run the function and averages time across all iterations. The benefit is we can be confident that the times reported are accurate and are not inflated or deflated due to variance.

Benchmark	Time	CPU	Iterations
SimpleScheduler Quicksort/64	0.018 ms	0.018 ms	41197
SimpleScheduler Quicksort/512	0.340 ms	0.339 ms	2063
SimpleScheduler Quicksort/2048	2.10 ms	2.09 ms	328
SimpleCSScheduler Quicksort/64	0.290 ms	0.197 ms	3412
SimpleCSScheduler Quicksort/512	3.89 ms	2.57 ms	274
SimpleCSScheduler Quicksort/2048	11.0 ms	7.25 ms	103
SimpleScheduler Fib/10	0.031 ms	0.031 ms	22742
SimpleScheduler Fib/15	0.346 ms	0.344 ms	2042
SimpleCSScheduler Fib/10	0.134 ms	0.092 ms	7405
SimpleCSScheduler Fib/15	0.512 ms	0.340 ms	2070
SimpleScheduler N-Queens/5	0.040 ms	0.039 ms	17783
SimpleScheduler N-Queens/8	2.03 ms	2.02 ms	347
SimpleScheduler N-Queens/10	42.4 ms	42.3 ms	17
SimpleCSScheduler N-Queens/5	0.083 ms	0.060 ms	10798
SimpleCSScheduler N-Queens/8	0.172 ms	0.122 ms	5716
SimpleCSScheduler N-Queens/10	0.280 ms	0.186 ms	3964

Concerning Issues

The main issue is whether the continuation stealing approach we have using setjmp/longjmp will actually work once we implement it with multiple threads. Although we have promising results with resuming execution, it is still a big unknown. Beyond that, it is just a matter of putting the work in, adding more tests, and finishing up our various implementations in time for benchmarking and creating the graphs.

Updated Schedule

Week	Goals

3/24 ✓	• Completed Proposal.
	• Conducted more Research on Implementation Strategies.
	  ▸ How does setjmp work and will we be able to use it for continuation stealing?
	  ▸ How exactly is OpenCilk implemented?–read research paper and documentation.
	  ▸ Figure out how function calls (especially recursive ones) are divided among the threads.
	• Begin to collect Cilk programs we want to benchmark.

3/31 ✓	• Created the simple scheduler, as described in work completed. • Worked on the simple child stealing scheduler as described in work completed. • Added fib, quicksort, n-queens, and rectmul tests.
4/7 ✓	• Finished the simple child stealing scheduler as described in work completed. • Began simple continuation scheduler. Works with one thread. • Began lock-free queue child stealing scheduler.
4/14	• Test out stack saving method for cooperative threads by Stephan Brennan (JACK) • Begin work on continuation stealing implementation (YONAH) • Work on lock-free implementation of Child Stealing scheduler (JACK)
4/21	• Complete continuation stealing implementation (YONAH/JACK helps if it is too difficult) • Complete lock-free and possibly other implementations of the child-stealing approach (different stealing strategies) and adapt it to the continuation stealing implementation for benchmarking (JACK/YONAH can help migrate the code over) • Add more benchmarking tests (JACK)
4/28	• Improve implementations with testing (BOTH) • Benchmark implementations, create tons of graphs, comparing speedup of all the implementations on a wide variety of benchmark programs. (BOTH) • Create poster displaying our results. (BOTH)