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

Yonah Goldberg
ygoldber@andrew.cmu.edu

Jack Ellinger
jellinge@andrew.cmu.edu

Project Web Page

Summary

We are going to implement a library to mimic the functionality of Cilk on top of C++. Users will be able to make calls to `cilk_spawn` and `cilk_sync`, on top of other potential Cilk primitives, and have our library manage concurrency.

Background

Cilk extends C++ with high level parallelism constructs. It allows programmers to easily write parallel code without low level thread management. Cilk implements fork-join parallelism. Users who want to fork off a function call to run in parallel can annotate their call `cilk_spawn my_function()`. When users want to join threads so that main thread execution can not proceed before all forked threads finish, they can insert a `cilk_sync` statement.

The benefit of fork-join parallelism is that Cilk programs behave almost exactly the same with Cilk primitives inserted as if they were not inserted. Therefore, it is extremely easy to parallelize a program quickly. Cilk programs also easily extend to divide-and-conquer algorithm implementations, where there are many recursive calls that can be forked to run in parallel. The goal of a Cilk programmer is to fork off enough units of work so that all worker threads stay busy. They want enough independent work to allow for good load balancing, but not too much independent work so that high granularity does not incur too much runtime overhead.

OpenCilk is implemented using a work-queue per-thread and a work-stealing policy. When a thread calls `cilk_spawn my_function()`, it indicates that `my_function` may be run in parallel. There are two strategies we could implement to allow for the parallelism:

1. Child Stealing - the function `my_function` is put onto the thread's work-queue and may be stolen by another thread to run in parallel. The main thread continues executing code beyond the function call.
2. Continuation Stealing - the main thread executes `my_function` and puts a continuation of the rest of the main thread's current function on the work-queue.

There are many interesting implementation decisions in Cilk that we want to explore. For example, the impact of child stealing versus continuation stealing, the impact of increasing thread count on performance, and the impact of different work-queue implementations.

The Challenge

The main challenge is how to provide efficient parallelization for a broad spectrum of use cases. We may make some implementation decisions that work well on some problems and poorly on others. It will be important to benchmark our implementation heavily to test the impact of our implementation choices.

The following are some decision tradeoffs that we intend to explore: child-stealing versus continuation passing; worker thread pool versus per-fork thread spawn; single work queue versus per-thread work queue. We can also explore various work-queue implementations and work-stealing strategies.

As a result, implementing Cilk is a very open ended problem, which allows us to test many parallel design tradeoffs we learned in class and see how they compare across different implementations workloads.

Resources

Since we are creating a run-time library, we are going to be starting from scratch, implementing everything in pure C++. For performance benchmarking, we will take many existing Cilk programs on a wide variety of applications and convert them to use our Cilk implementation(s). We will want to test our implementation strategies on a wide variety of problems that benefit from different thread counts. We would benefit from access to the Pittsburgh Supercomputing Center machines to test with more threads.

For references on how to start with our implementation, we have the following:

- 15-418 lecture slides on the implementation.
- The official original Cilk research paper from MIT
- The official Cilk documentation.

Goals and Deliverables

What we Plan to Achieve

We plan to implement two versions of the Cilk API. One version is based on simply spawning a pthread for `cilk_spawn` and joining pthreads for `cilk_sync`. The other version mirrors the OpenCilk implementation to the best of our ability, using continuation stealing with setjmps and per-thread work queues.

We plan to benchmark our implementations on a wide variety of test inputs against the OpenCilk implementation, both on GHC at low thread counts and on PSC at higher thread counts.

What we Hope to Achieve

We hope to explore many implementation decisions for our parallel library. For example, it would be interesting to benchmark child stealing versus continuation stealing or different thread safe work-queue implementations. Essentially, whatever time we have left after meeting our original goals, we will spend testing small implementation tweaks, comparing performance, and trying to come up with the best possible implementation on our benchmark suite given the time constraints.

As a final fun challenge, if we end up with a lot of extra time, we will attempt to create a simple compiler for our best API implementation that mimics the OpenCilk fork-join syntax.

Goals if Work Progresses Slowly:

If implementing continuation stealing with setjmps and per-thread work queues turns out to be too challenging, we can relax the difficulty of our implementation and try something easier. For example, child stealing might be easier to implement because we know it will be easy to store lambdas/function pointers on the work-queue. Another relaxation might be only implementing one shared work-queue that all threads pull from.

Demo/Poster Description:

Our poster will 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.

Questions we Plan to Answer and What we Hope to Learn:

We plan to answer what the implementation tradeoffs are when designing parallel runtimes. What strategies work the best and on which types of problems? When does our simple implementation work best and at what point does it fail? What are the benefits of continuation stealing versus child stealing and why? How do our implementation decisions affect load balancing and cache misses?

We hope to achieve a greater understanding of how a widely used parallel runtime is implemented and why certain implementation choices are made. We also hope to apply parallel design tradeoffs we learned in class and learn their effects on a real application.

Platform Choice

We are choosing C++ for this project because it provides mechanisms for low-level memory and thread management. It has a lot of concurrency support allowing us to try various different synchronization strategies that we learned in class. It is also highly performant, and since we are looking to create a framework to make high performance computation easy, this is a necessity.

Schedule

Week	**Goals**
3/24	• Complete Proposal. • Conduct 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	• Create Cilk API using p_threads to begin benchmarking strategies ‣ Use pthread_create for cilk_spawn and pthread_join for cilk_sync. • Begin thread pool implementation with one thread-safe work queue ‣ cilk_spawn pushes to the work queue, either a function pointer in the case of child stealing or a continuation in the case of continuation stealing. ‣ Initialize a thread pool and all threads steal work from the single queue. • Finish collecting Cilk programs we want to benchmark.
4/7	• Complete thread pool implementation with one thread-safe work queue, either with child stealing or continuation stealing. • Begin thread pool implementation with one work-queue per thread and the more complex stealing implementation.
4/14	• Continue work on complex Cilk implementation.

4/21	• Finish complex Cilk implementation. • If we implemented child stealing, now implement continuation stealing. If we implemented continuation stealing, now implement child stealing. • Potentially create compiler that mimics Cilk syntax. • Potentially try out more interesting implementation decisions, such as work-queue implementation (mutexes versus lock free).
4/28	• Benchmark implementations, create tons of graphs, comparing speedup of all the implementations on a wide variety of benchmark programs. • Create poster displaying our results.