Lab 6 Report

Group: Spider-Man
Members: Joseph Li (jxli) and Konwoo Kim (konwook)
Fall 2021

Introduction

For our project, we've implemented Milk, a threading runtime system similar to Cilk. Our L6 compiler supports general task parallelism through milk_spawn and milk_sync calls as well as the following requirements: automated work stealing, correct synchronization of child subroutines, behavior preservation of serial programs, and a speedup for parallelizable code. Our main contributions are a runtime library implemented in C [`run411milk.c`] and modifications to our L5 compiler. We find that we get a speedup on the majority of tests which have significant room for task parallelism.

Project Specification

We first discuss the grammar and semantics of our two new keywords: milk_spawn and milk_sync. The grammar rules are: milk_spawn **ident** <param_list>; and milk_sync;. Milk_spawn can only be applied to a function call while milk_sync is a standalone statement.

No additional computation can occur between the last milk_spawn and matching milk_sync, and computation halts at a milk_sync until all matching milk_spawn calls finish.

The metrics for success for this project were to modify our compiler and implement a runtime system which would support:

- Automated work stealing
- Correct synchronization of child subroutines
- Behavior preservation of serial programs
- Ideally, a speedup for parallelizable code
- Correct behavior on our L6 tests and backwards compatibility with tests from L1-L5

Implementation

Because we focused on implementing the runtime system for this project (which involved the core functionality), we weren't able to finish modifying the frontend of our compiler. Instead, we manually elaborate the logic using functions from the runtime library when writing the test cases. An example is shown below for a test case which computes the nth Fibonacci number.

```
int fib(int n) {
  if(n == 0 || n == 1) {
    return 1;
  }
  int left = milk_spawn fib(n - 1);
  int right = milk_spawn fib(n - 2);
  milk_sync;
  return left + right;
}
```

The usage of milk_spawn and milk_sync in the above code block follows our spec from above. Elaboration involves three main steps:

- **Initial setup** of allocating space for return values, creating the milk stack frame and entering it
- **Spawning the actual recursive calls** using setjmp
- **Synchronizing the results** and cleaning up pointers

```
int fib(int n) {
    if(n == 0 || n == 1) {
        return 1;
    }

    // BEGIN MILK ELABORATION

    int *__MILK_RES_left = malloc(sizeof(int));
    int *__MILK_RES_right = malloc(sizeof(int));

    int *__MILK_SF = milk_create_frame(2);
    int *__MILK_SF_CTX = milk_extract_context(__MILK_SF);
    milk_enter_frame(__MILK_SF);

    if(__builtin_setjmp(__MILK_SF_CTX) == 0) {
```

```
        spawn_fib(n - 1, __MILK_RES_left);
    }

    if(__builtin_setjmp(__MILK_SF_CTX) == 0) {
        spawn_fib(n - 2, __MILK_RES_right);
    }

    // sync
    int left = *__MILK_RES_left;
    int right = *__MILK_RES_right;

    free(__MILK_SF);
    free(__MILK_RES_left);
    free(__MILK_RES_right);

    // END MILK ELABORATION
    return left + right;
}

void spawn_fib(int n, int *res) {
    milk_increment_sf_spawned();
    milk_push_frame();
    *res = fib(n);
    milk_exit_frame();
    return;
}
```

We implement a change to our compiler to support ___builtin_setjmp instruction. In essence, this instruction saves a%rip, %rsp, and %rbp into a buffer that can be used to reload the current stack situation. To implement this, we have a special handle for function calls to ___builtin_setjmp that customizes its use-def (it clobbers every single register). In addition, this function is inlined to get an accurate snapshot of the current %rsp. Implementation for this function is found in target.rs and emit.rs.

Now, we discuss the runtime system in detail. We split our discussion into three main sections: an overview of the design, main data structures, and the runtime dynamics.

Milk Design

This implementation is inspired by Cilk. There are two sets of “realized data structures”: the milk call tree and milk deques.

- **Milk Call Tree:**
    - Given a full call tree, some subset of this tree describes those calls that were done using a milk_spawn.
    - The Milk Call Tree is formed by the structure of the parent milk_sf pointers, where the children point to the parent (reversed pointer direction from a typical tree structure).
        - For fibonacci, this forms some binary tree.
            - Each milk region has two milk_spawns.
        - Naturally, for tribonacci, this forms some 3-nary tree, and so on.
        - If you for some reason decide to only spawn one child in your milk region (i.e. no parallelism, at least in our runtime), this is a singly linked list from the last spawn to the first spawn.
    - Each worker holds a current_sf milk_sf pointer that points to some location on this milk call tree.
    - This call tree is heap-allocated and is created/destroyed as milk sections go in and out of scope.
- **Milk Stealing Deques:**
    - Each worker holds a deque of milk_sf pointers. This represents the possible continuations that a different worker can steal.
    - Thieves try to pop off continuations from the head of the deque (oldest spawns).
    - The worker that holds this deque manages the tail of the deque.
        - It constantly looks at the tail of the deque for more work.
        - If the worker spawns some process or steals some continuation, then it pushes it to the tail of the deque.
            - It only does not do this if that was the last continuation for that milk region (i.e. target == spawned).
            - For example, if a milk region has two spawns and the worker performs the second spawn, then there is no more work to steal.

Milk Data Structures

Global Variable: gstate (struct milk_state): Holds current state of milk runtime.

Thread-Local Globally-Accessible Variable: workerid (___thread int): It is one of 0..nworkers-1.

C-Structures:

- **struct milk_worker**
    - **Deque-Related:**

        - deque (milk_sf *arr[N]): Holds a deque of always stealable stack-frames (i.e. number of spawns < target)
        - deque_head (int): Index of front of deque (inclusive)
        - deque_tail (int): Index of back of deque (exclusive)
        - deque_max_size (int): max size the deque can grow to
        - lock (pthread_mutex_t): Access lock for the deque
    - **General Status Related**
        - id (int): Worker index. usually one of 1..(num_processes - 1)
        - current_sf (milk_sf *): The most recent milk_sf when going up through the call tree.
    - **Worker Loop Related:**
        - loop_env (void *arr[5]): Holds some register variables so you can jump back into the worker_loop and begin to steal.
        - edit_env (void *arr[5]): Deprecated.
        - safe_rsp (void *): Holds a stack pointer where you can safely grow a new arm of the cactus stack.
- **struct milk_sf**
    - env (void *arr[5]): Holds some register variables so you can jump back into the milk section related to this frame.
    - parent (milk_sf *): Pointer to the closest milk_sf that is
    - owner (int): Index of the worker that created this milk_sf.
    - true_rsp (void *): Stores the actual value of %rsp. This is used to reinstate the value of %rsp as it might have been clobbered by thieves using their cactus stack.
    - target (int): Total number of spawns that are related with this frame.
    - returned (atomic_int): Of the number of possible spawns, how many have returned.
    - spawned (atomic_int): Of the number of possible spawns, how many have been spawned.
- **struct milk_state**
    - nworkers (int): number of workers
    - workers (struct milk_worker *): Pointer to each worker struct.
    - threads (pthread_t): Handle for each spawned worker thread.
    - exit (int): 1 if we want workers to exit (they read this variable when trying to steal), 0 otherwise.
    - verbose (int): Used for debugging.

Milk Dynamics

- Running _co_main:

    - Start all pthreads on worker_loop. Worker with workerid == 0 runs `__c0__main`. This worker will be responsible for returning the value and initiating the cleanup process.
- Worker Looking For Work:
    - Stealing
        - Loop through list of workers (except yourself)
        - Acquire a lock on the deque
        - If the deque is non-empty, pop the head element. Modify the continuation/jump buffer to use the current RSP to create the cactus stack.
        - Jump to this edited continuation.
        - Release lock on the deque after attempting to push the head element back (might fail if the stack_frame is exhausted of continuations). This is done after moving forward the setjmp location of the continuation.
    - After a Return
        - If you are the owner of the current milk_sf, then you are responsible for seeing the function through its true return.
            - If everything has been returned, restore the correct %rsp and longjmp back to the last continuation and resume the function.
            - If not everything has been returned, but everything has been spawned, then wait in a loop for the other threads to return.
            - If not everything has been spawned, take the next continuation and spawn some work.
        - Otherwise, you were a thief and this is the bottom of your cactus stack, so you jump back to the worker_loop.

Testing Methodology

Our tests for our compiler are under `lab6/tests`. They were designed to test the correctness and performance of our compiler and cover applying spawn and sync for recursive problems, having 2 spawn calls in a single region, having an arbitrary number of spawn calls in a single region, and various stress tests for looking at speedup.

Our test cases look at problems which can benefit from task and data parallelism like range queries on top-down segment trees or calculating matrix determinants. Half of

the test cases are written from scratch while the other half are modified from `l4-large`.
Tests can be run using `./grade_tests.sh tests/` which takes in the directory as input and outputs the results of the tests. We also test our compiler on L4-large and it is backwards compatible.

Analysis

Our code is correct on all of our tests, and the figures show the ratio of speedup using 4 threads vs 1 thread with our implementation. We find that on the majority of test cases which have a large # of recursive calls (like det.c, fib.c, ld.c, choose.c) we have a nontrivial speedup. Interestingly enough, the test case with the top-down segment tree range queries, we are significantly slower than using 1 thread. This could be due to unbalanced workloads and a general low amount of arithmetic intensity. This results in a lot of runtime overhead as it's not doing much useful work.

A clear future improvement for our compiler is to change the frontend for easy integration instead of manual elaboration of milk_spawn and milk_sync. This would be implemented by identifying milk regions, verifying they do not do dependent computation, then following the elaboration steps to create spawn_* versions of spawned functions and to create/free memory locations for stored return values.