Introduction

We have implemented C-SPAN, a language that adds support for parallelism to L4 via the introduction of four basic primitives: spawning concurrent function calls, waiting for calls to complete, semaphore operations, and atomic integers. These primitives allow programmers to implement higher level constructs such as mutexes, condition variables, spin-locks, lock-free linked lists, and parallel algorithms.

Specification

New Tokens/Keywords

- spawn
- wait
- future
- semaphore
- atomic
- cmpxchg

New Grammar Rules

- <type> ::= **semaphore** | **atomic** | **future<** <type> **>** | …
- <exp> ::= **wait** <exp> | **spawn ident** <arg list> |
  cmpxchg(<lvalue>, <exp>, <exp>**)** | …

Static Semantics

- Atomics, futures, and semaphores are small types.
- Atomics/semaphores in local variables must be initialized at declaration. Atomics/semaphores that are allocated in heap memory are initialized to 0 by default.
- Futures allocated in heap memory are initialized to a null value.
- Semaphores always represent non-negative integers and can only be used in postOp <simp>'s.
- Atomics are integers that can only be modified through the following assignment ops, (+=, -=, &=, |=, ^=).
- The atomic and int types are implicitly cast to each other in expressions, however struct/array/pointer/future types using them are considered distinct, so therefore int[] != atomic[].
- The spawn keyword may be applied to any function call expression. Its type evaluates to a future<T>, where T is the return type of the function call.
- The wait keyword may be applied to any expression of future type. If the type of exp is future<T>, then wait exp has type T.
- Wait <exp> cannot be of type void, unless used in an expression clause of a <simp>
- Spawn and wait are unary operators with equal precedence to the other L4 unary operators. Spawn may only be applied to function call expressions.

- The cmpxchg intrinsic function has type (T, T, T) -> T where T is any small type. Its first argument must be an lvalue.

Dynamic Semantics

- A spawn expression will return a future value immediately upon evaluation, and the referenced function will be queued for concurrent execution. It will complete at some point in the future.
    - Each spawn expression allocates a new thread and future. If these allocations fail, a safe memory error is signaled.
- The wait keyword ‘unwraps’ the return value of a concurrently executing function by waiting for its execution to complete. A wait expression is not evaluated until the function call referenced by the future completes.
    - Waiting on a future that has not been initialized to a spawn expression signals a safe memory error.
    - If wait is called on a future that has already been waited upon and completed, a safe memory error is signaled.
    - If multiple threads call wait on the same future, the behavior is undefined.
- The ++ operator increments semaphores by 1. The -- operator will decrement a semaphore if it is positive, otherwise it will cause the calling thread to wait until the semaphore is positive and then decrement it.
    - The first time a semaphore is initialized or acted upon, it will also allocate a semaphore type. If this fails, a safe memory error is signaled.
- Atomics that are modified in multiple concurrently executing functions will have a sequentially consistent result after all relevant functions return.
- Cmpxchg compares the value of its first and second argument, and if they are the same, sets the lvalue argument to the value of the third argument and returns the second argument. If they differ, the lvalue is not updated and it returns the compared value of the lvalue. This process is carried out atomically, so no other threads may modify the lvalue in between the compare and swap operations.

Implementation

Lexing/Parsing

Lexing was fairly straightforward for this lab, just adding the 6 new keywords introduced above. The biggest issue was the use of <, > for the future type, which makes it harder to differentiate between >, >>, >>= tokens. The lexer now stores a future depth counter, of how many futures it parses, and decrements the counter upon parsing a > token. This causes the lexer to not be context-free, since it will lex a given string differently based on what it has seen beforehand. This is still fine since it will lex the tokens correctly if the entire program is well-formatted. Our recursive descent parser didn't change much in this lab. The spawn/wait expressions were just added to the unary phase of expression matching, and parsing a future type is apparent from the leading “future” token.

Type Checking

There were three new types added to this lab. The future type is the most unique due to its polymorphic ability to create a future of any type. Since functions are statically typed, this allows us to still determine a unique type for every spawn expression. These types are also very similar to the array/pointer types, so a similar use of an implicit type graph was used. Due to the decision of allowing implicit atomic/int casting in expressions, most of the binary ops are now overloaded to accept either type, but will return an int type as a result, since (x+y) cannot be atomically written to. Due to the new semantic meaning of ++,-- on post ops, desugaring x++ => x += 1, is now incorrect if x : semaphore. This covers most of the significant changes, minor changes were made to obvious differences in semantics, such as enforcing all three arguments to cmpxchg are small and of the same type.

Codegen

Our code generation algorithm did not fundamentally change with the addition of C-SPAN features, but we did have to add new abstract instructions for spawn, wait, cmpxchg, and operations on non-local atomics. We changed our lvalue munching algorithm to emit fused operations (binops & cmpxchg) for lvalues describing memory locations, which was necessary to support emitting atomic instructions instead of deref-op-store chains (which, incidentally, we realized we forgot to optimize in L5). We also added a flag to each temporary that indicates if it needs to be atomic.

Backend

Our x86 backend also did not fundamentally change, but was modified to support atomic temporaries and the new abstract instructions. Operations on atomic temps now output x86

arithmetic instructions with the ‘lock’ prefix. However, this is unnecessary when the destination is a register, so we only do so when the destination is a memory location (which is fused for atomic stores by codegen). Cmpxchg is implemented very similarly using the locking cmpxchg instruction. Both areas still support null checking their operands for safety. Threading and semaphore operations are emitted as calls to our runtime.

Runtime

We implemented a small runtime (**runCSPAN.c, runCSPAN.s**) for supporting the thread and semaphore based operations that relies on the pthreads library. Our compiler will emit calls to the runtime for spawn, wait, and semaphore instructions.

The spawn operation allocates one of our future values, which contains a pthreads thread identifier, a pointer to the C-SPAN function to run, and a variable-length array of parameters appended to the struct. Our runtime function is implemented as a C vararg function, which the compiler backend respects. A new thread is then created to run our thread initialization function, which takes the future, sets up the parameters in the proper registers (or copies them to the stack if necessary), and calls the C-SPAN function. Setting up the stack required some assembly implementation in **runCSPAN.s**. The spawn operation then returns the allocated future as a pointer. The wait operation simply takes a pointer returned from spawn and uses pthread’s join operation to get its return value.

The semaphore operations are structured similarly, except that unlike a future that can simply be NULL, zero-initializing a semaphore requires a runtime call to allocate a new semaphore type. This meant that when allocating arrays or pointers to semaphores, we had to make them semantically initialized to zero without emitting calls to initialize the heap memory. To achieve this, we made our runtime semaphore operation function detect non-local un-allocated semaphores and automatically zero-initialize them. However, doing so introduced a race condition between multiple threads initializing the same semaphore, so we made use of the C11 cmpxchg function to remove the race condition (only one thread gets to zero-initialize the semaphore). This can leak memory, but the “garbage collector” will deal with it.

Finally, note that our backend assumes the first second parameters of the vararg spawn function are still passed in registers, which requires the runtime to be compiled with optimizations enabled.

Correctness Tests

We wrote a suite of C-SPAN specific tests that can be found in the **lab6/tests** folder. They cover the following areas:

- Passing >6 parameters to spawned functions
- Atomics are truly atomic and have null checking
- Memory errors when running out of resources
- cmpxchg is atomic and has null checking

- Nested/polymorphic futures and type checking them
- Various parsing precedence issues
- Semaphores work properly and don’t have an init race
- Type checking casting between atomics and ints
- Type checking void wait expressions
- Type checking typedefs with futures and structs
- Memory errors on multiple waits & uninitialized futures
- Post-ops still working

Unfortunately, we didn’t have the testing infrastructure to exhaustively test correct waiting/blocking behavior, but we believe everything to work properly since we rely on pthreads to implement correct concurrency primitives, and our user library tests all work as expected.

Tests may be run with the **grade_tests.sh** script in the **lab6/** directory and takes in the directory where the tests are contained as an argument, which will compile our runtime and tests, emitting executables in a run/ folder. It doesn’t autograde the tests, but will print the result of each test. All tests pass.

To ensure correctness, we also regression tested our compiler against all the tests from L1-4. Our compiler succeeds on all tests except the following, which fail due to using ‘wait’ as an identifier:

- l3-large/voldemort-spells.l3
- l3-large/squirtle-return-typedef-prop-correct.l3
- l4-large/alliser-potato_04_hachiko.l4

User Library Tests

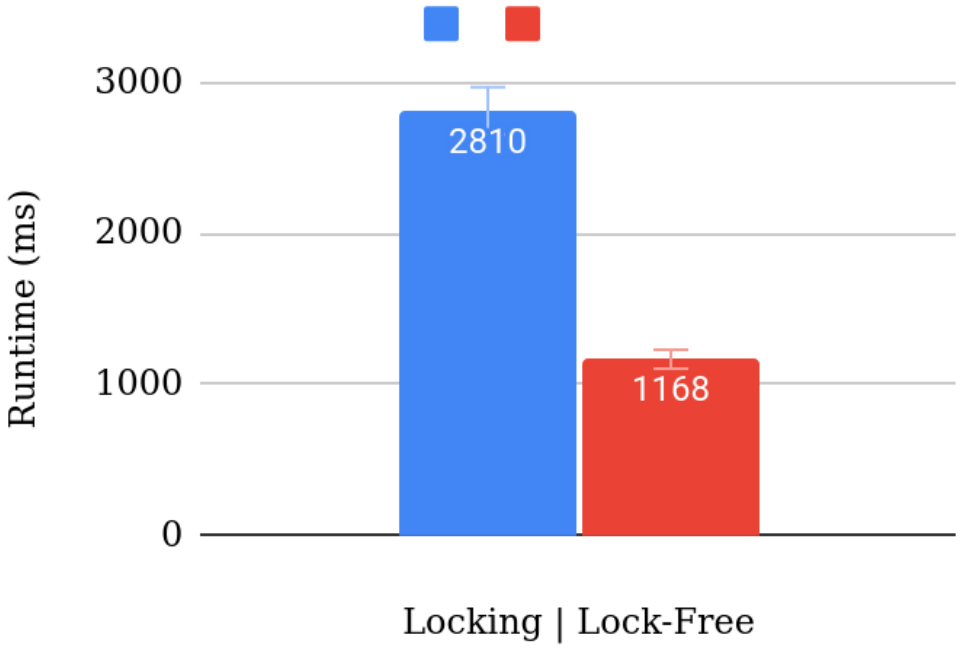
We implemented a variety of higher-level concurrent constructs to demonstrate that our primitives are sufficient for doing real work with parallelism. These tests/examples can be found in **lab6/tests_lib**.

- **mutex.cspan**
  - Uses semaphore primitive to implement a very simple mutual exclusion lock that will block when waiting for an unlock.
- **spinlock.cspan**
  - Uses atomics and cmpxchg to implement a ticket-based mutual exclusion lock that busy loops when waiting for an unlock.
- **cvar.cspan**
  - Uses semaphore primitive to implement a simple condition variable, which wraps a lock and provides functionality for the standard condition wait and signal operations. It is also possible to support the broadcast operation, but we did not.
- **Linked List**
  - **linked_list_l.cspan:** Uses semaphore primitive to implement a simple linked list with fine-grained locking for insertions.
  - **linked_list_lf.cspan:** Uses cmpxchg primitive to implement a simple linked list with lock-free concurrent insertions.
- **Merge sort**
  - **mergesort_seq.cspan:** Implements a standard sequential merge sort with 2 arrays, using O(n log n) work and span.
  - **mergesort_par.cspan:** Implements a parallel merge operation that uses O(n) work but $O(\log^2 n)$ span, resulting in $O(\log^3 n)$ span overall with the same work complexity.
- **Matrix Multiply**
  - **matrix_seq.cspan:** Computes a matrix product sequentially. $O(n^3)$ span/work
  - **matrix_seq.cspan:** Computes a matrix product in parallel by blocking the output matrix so each thread computes different rows of the output $O(n^2)$ span now
- **Map-Reduce**
  - **mapreduce_seq.cspan**: sequentially maps and reduces an integer array
  - **mapreduce_par0.cspan**: maps and reduces an integer array by spawning a thread for each block of elements and using a single atomic for the reduce accumulator.
  - **mapreduce_par1.cspan**: maps and reduces an integer array by spawning a thread for each block of elements and then using a parallel divide-and-conquer reduce.

Analysis

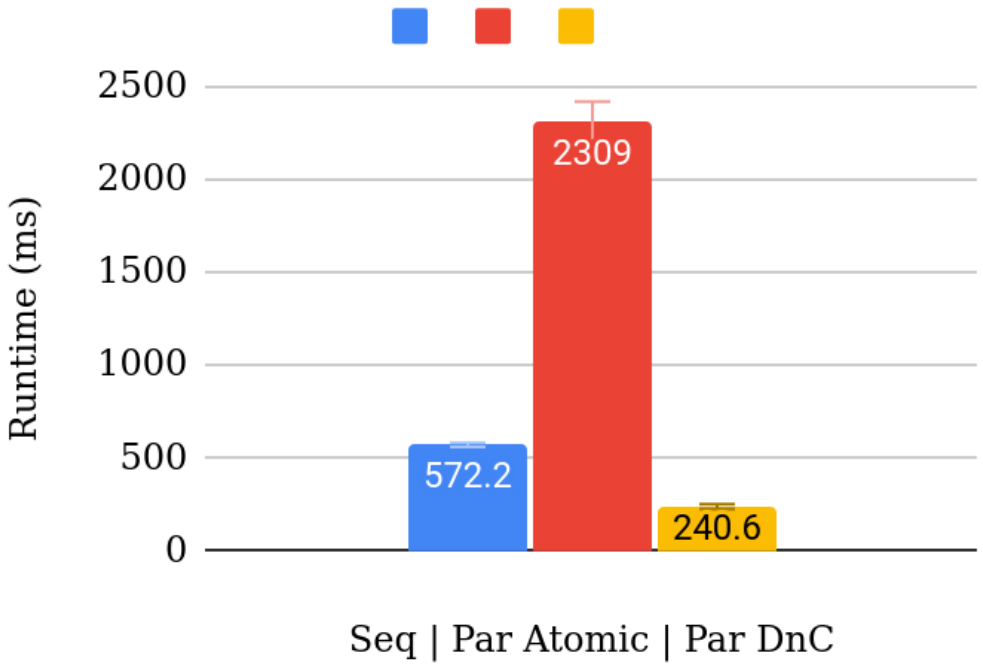
We benchmarked several versions of our mapreduce test, our parallel merge sort test, and our parallel matrix multiply test to demonstrate speedup that comes from our parallel implementations. All tests were run in unsafe mode with GCC optimizations (O2) enabled for the runtime C code on a 4 core CPU without hyperthreading. All runtimes were averaged over 10 runs or more as directed by the hyperfine profiling tool.

Locking vs. lock-free linked list insertions (1000 insertions on 10000 threads):



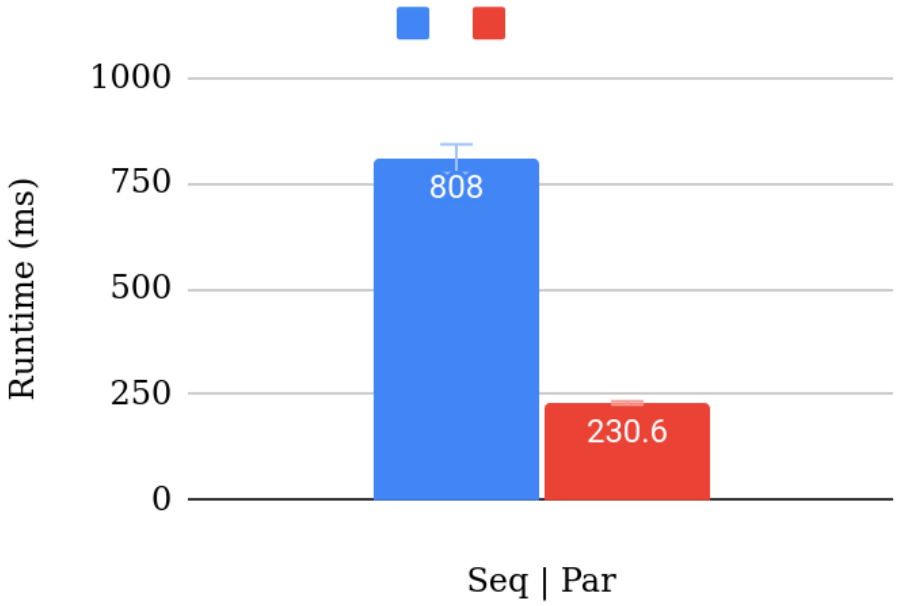
This result was expected, as going through the runtime to lock and unlock a semaphore-based lock for each insertion is much slower than using the atomic cmpxchg operation.

Map-Reduce: sequential, parallel with all threads reducing to an atomic, and parallel with divide-and-conquer reduction. (100000000 integers, 1000000 grain size)



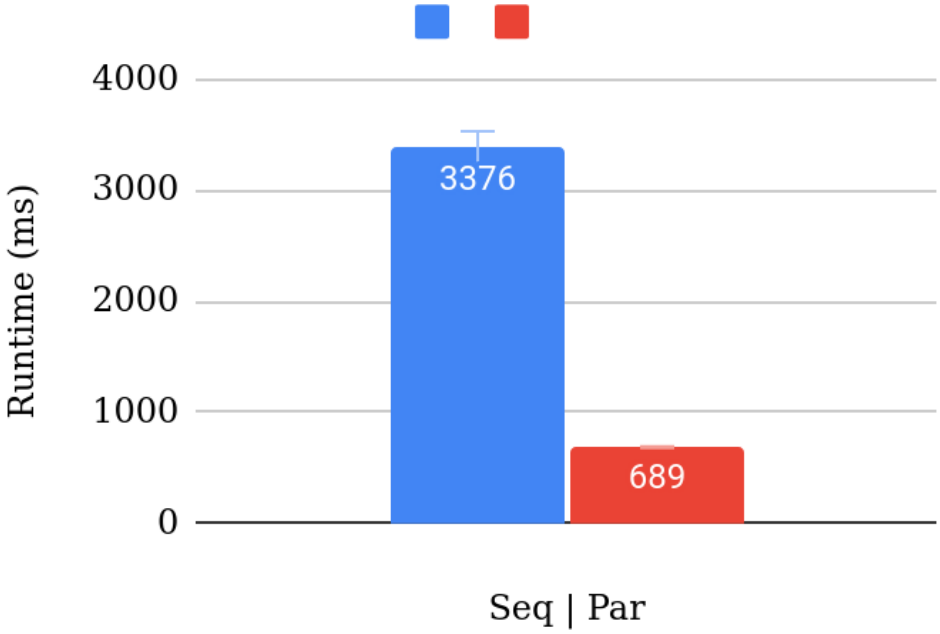
This result was more interesting, as it turned out that having threads fight over a single accumulator, even a lock-free atomic one, massively outweighed the speedup compared to the sequential implementation. The divide-and-conquer parallel reduce was however over twice the speed of sequential, due to each thread having a dedicated accumulator. This was despite the overhead of spawning a full tree of threads.

Mergesort: sequential and parallel. (2^{22} size, 100000 grain size):



This parallel result was nearly four times faster than the sequential result.

Matrix Mul: sequential and parallel. (1024 size, 100 block size)



Interestingly, the parallel version was over 4 times faster than the sequential one on a 4 core CPU. We hypothesize that this was due to improved caching effectiveness when only dealing with one block of the matrix at a time and having access to additional L1/L2 caches.

Future Work

C-SPAN is currently limited due to the use of OS-provided threads to represent each concurrent function call. This means that the runtime can only spawn a couple tens of thousands of threads before running out of resources. Further, the runtime and programmer have little control over the execution schedule of the various threads.

To remedy these issues, we could implement our own user-space M:N threading system and scheduler, i.e. implement coroutines as seen in languages like Go. This would allow us to support features like priorities and yielding, as well as simplifying inter-thread communication. Mapping arbitrary coroutines to a constant number of OS threads would also allow for the creation of potentially huge numbers of concurrent coroutines, as well as optimizations like not allocating a stack for each coroutine.

Another issue with the grammar that came up when writing benchmark tests is that the syntax in CSPAN for spawning an arbitrary number of threads is clunky. It requires allocating a future<T>[], a for loop of spawning them, and then a subsequent for loop for waiting on all of them. Future additions for the grammar could be a parallel for loop of the form **parfor(i, n) spawn foo(...)**, that given a precalculated n, spawns the threads in fashion as stated above.