Comparing Producer-Consumer Implementations in Go, Rust, and C

GABBI FISHER*, Stanford University, U.S.A.
CHRISTOPHER YEH*, Stanford University, U.S.A.

Additional Key Words and Phrases: Producer, Consumer, Buffered Queues, Channels, Multithreading

ACM Reference Format:
Gabbi Fisher and Christopher Yeh. 2017. Comparing Producer-Consumer Implementations in Go, Rust, and C. 1, 1 (December 2017), 7 pages. https://doi.org/

1 INTRODUCTION

The producer-consumer model is a classic synchronization problem, exploring the division of labor between threads. Producers are responsible for creating or yielding items of some kind, while consumers are responsible for processing these items. The producer-consumer model is present in a wide array of multithreaded applications. It is commonly used to read data from high-throughput streaming APIs, and underlies the design of popular streaming data services such as RabbitMQ and Apache Kafka. Producer-consumer is a model frequently used in other data-intensive tooling, frequently appearing in any tooling used to ingest and process large quantities of data.

Given the prevalence of the producer-consumer model, we sought to examine implementations of this model across systems programming languages. The producer-consumer model offered a valuable opportunity to explore and compare concurrency tools among systems languages, and implementing producer-consumer in multiple languages could reveal the utility of using certain languages for data-intensive processing. In our research, we seek to answer two questions: (1) how did concurrency tools such as threads and shared/concurrent queues broadly differ across systems languages and (2) how did ease of implementation and performance compare between these languages?

In this paper, we implement and benchmark producer-consumer models in the increasingly prominent systems languages Go and Rust, in addition to the mainstay language C. We analyze the ease of writing multithreaded code in these languages, and the subsequent performance of our implementations on common producer-consumer use cases. Our code can be found in our GitHub repo at https://github.com/gabbifish/producer-consumer.

2 A BRIEF OVERVIEW OF THE PRODUCER-CONSUMER MODEL

The producer-consumer model consists of two pools of threads, unsurprisingly called producers and consumers. All of the threads have access to a shared queue.

Producer threads are responsible for ingesting (and possibly preprocessing) data. When a producer reads in data, it inserts it into a shared queue. Consumer threads read in data from the queue for processing, potentially yielding the data for further action or storage.

*Equal work was performed by both project members.

Authors' addresses: Gabbi Fisher, Stanford University, P.O. Box 12383, Stanford, CA, 94309, U.S.A., gsfisher@stanford.edu; Christopher Yeh, Stanford University, P.O. Box 16135, Stanford, CA, 94309, U.S.A., chrisyeh@stanford.edu.

2017. Manuscript submitted to ACM

This queue is a buffered queue; the writer does not block if a reader has not read the value they most recently wrote. Writers simply write to the buffer if it has space, and only block when the buffer's size has reached capacity. Readers only block when the buffer is empty. As such, producers write to the shared queue while it is not full, and consumers read from the shared queue when it is not empty.

When all producers have finished ingesting data and writing it to the buffer, they close the buffer, prohibiting further writes to it. The producer threads then die. Consumers are still free to read from the buffered queue until it is both empty and closed, after which the consumers threads complete their final data processing and also die.

Sections 3, 4 and 5 of this paper analyze how threads and buffered queues are implemented in Go, Rust, and C. These sections delve into the design differences between concurrency constructs in these languages.

To actually test where these concurrency constructs could play a role in the producer-consumer model, we built four implementations–one in Go, one in Rust, and two in C. Building producer-consumer in each of these languages allows us to compare concurrency tooling in action, in addition to benchmarking the performance of producer-consumer models across languages (Section 6). Through implementation and analysis, we explore the utility of newer systems languages in a frequently used multithreading application.

3 IMPLEMENTATION IN GO

Go is a programming language created by Google in 2009, designed to offer safety and simplicity over systems languages like C. We were compelled to examine producer-consumer in Go given Go's increasing popularity.

Go is especially well-known for two concurrency constructs: goroutines and go channels. Goroutines are utilized to create threadpools, while Go channels are equivalent to asynchronous buffers. As such, they both play a large role in how producer-consumer is implemented in Go.

3.1 Goroutines

Goroutines choose to minimize operations in the kernel space to produce a lightweight thread. Minimizing operations in the kernel space is a substantial optimization, given that context switches between kernel and user space are expensive. (Context switches require saving the process's set of registers and restoring another process from another saved set of registers). Goroutines minimize context switches via the following mechanisms:

(1) Goroutines are multiplexed onto a threadpool of kernel threads.
(2) Goroutines are scheduled by the Go runtime, not the operating system.
(3) Goroutines yield to others at predefined points, so switches between goroutines are constrained to a limited set of calls.

Multiplexing goroutines onto a preexisting thread pool ensures that no new operating system threads have to be created or joined. The creation and joining of kernel threads involves requesting resources from the operating system and returning them when the thread has finished. Both of these actions are expensive processes requiring context switching. Instead, Go maintains a pool of OS threads it can schedule goroutines onto, negating the need to spawn a new OS thread for every new goroutine. [6]

Go schedules goroutines onto OS threads in the Go runtime, again preventing expensive calls to the OS scheduler. Go also eschews preemptive scheduling in favor of cooperative scheduling. Preemptive scheduling allows the OS to intervene with threads and forcibly suspend them; this requires additional overhead and requires kernel access. Cooperative scheduling has threads yield to each other at predefined points in execution. Cooperative scheduling can

either be handled by the OS or within the user space. Consistent with its effort to constrain thread management to the user space, Go implements cooperative scheduling in its runtime. Goroutines yield to others in places such as writing to or receiving from channels, in addition to performing blocking syscalls.[3]

Starting a goroutine is as simple as calling a function `funcName(arg str)` and passing in arguments: `go funcName("argument")`. The goroutine dies when it reaches the returns from or reaches the end of the provided function.

3.2 Go Channels

Go channels are a queue shared across threads. They can work as unbuffered channels (writers block until their value is received) or buffered channels (as described in section 2).[7] For our purposes, go channels offered an out-of-the-box implementation of the buffered queue necessary for producer-consumer.

Initializing a buffered queue is simple: `intQueue := make(chan int, 100)` where 100 is the queue size. Writing to the channel is done as such: `intQueue<-newVal` where newVal is an integer. Reading from the channel is equally simple: `readInt := <-intQueue`. Closing a channel to new writes is done through calling `close(intQueue)`.

4 IMPLEMENTATION IN RUST

Rust is newer systems programming language originally developed in 2010. Intentionally designed to help programmers write safe, concurrent code, Rust was a natural pick for a language in which to implement the producer-consumer model.

4.1 Rust Threads

Threads in Rust are more similar to pthreads in C, than to goroutines in Go. In order to provide a smaller runtime, Rust's standard library only provides an implementation of 1:1 threading, so a single thread in Rust corresponds to a single kernel thread. It defers the Go-style M:N multi-plexing threading model to crates outside of the standard library. Furthermore, while Rust does provide a `std::thread::yield_now()` function in its standard library for implementing cooperative thread scheduling, the `chan` crate does not take advantage of this primitive and instead relies on the operating system's thread scheduler.

4.2 Rust Channels

In its standard library, Rust provides a built-in multiple-producer single-consumer FIFO queue as part of the `std::sync::mpsc` module. However, since this module only supports one consumer, we decided against using it. Rust also provides synchronization primitives such as mutexes and condition variables. However, we decided to use the popular crate `chan` instead, as it provides easy creation of Go-like multiple-producer / multiple-consumer channels, and it does not use any `unsafe` code.

Behind the scenes, `chan` creates threads with shared access to a vector that serves as the buffer, which is protected by mutexes and condition variables. From However, the creator of `chan` suggests that by avoiding `unsafe` code, `chan` incurs a performance penalty. "Performance takes a second seat to semantics and ergonomics," he writes in the README for the `chan` GitHub repo[4].

5 IMPLEMENTATION IN C

C is the classic systems programming language. Its default provided threads are in the kernel space, while it offers no library implementation of a buffered queue–leaving its design up to the programmer.

Because C is such a mature language, there are robust open-source libraries for threading in user space. There is even a Go-style concurrency library for C. We additionally examine both of these libraries, looking for additional support and performance optimizations in implementing producer-consumer.

5.1 Pthreads

Pthreads, otherwise known as POSIX threads, are the threads most commonly used in C. They are launched, preemptively scheduled, and joined by the kernel, meaning a series of expensive context switches from userspace are necessary. Like Rust's kernel-based threads, the usage of kernel-scheduled threads stands in contrast to Go's utilization of userspace-scheduled threads.

Spawning a pthread is not difficult with the pthreads API, but these threads require the explicit use of mutexes and condition variables to preserve memory safety while working a buffered queue. Threads must also be explicitly joined. Using pthreads is a far more involved task than simply spawning a goroutine that handles its own joins.

C also offers no buffered queue structure; instead, we had to implement one. Our buffered queue is represented by a struct that stores a capacity, a current size, and a C array that is filled and drained like a FIFO queue. Unlike Go or Rust channels, this buffered queue does not come with memory safety guarantees. This buffered queue struct, then, must also include mutexes and condition variables for signaling when it is safe for producers to write and consumers to receive. Additional logic is also necessary for ensuring that new allocations and reads from the buffer "wrap around" when they reach the end of the array allocated for the buffer.

5.2 Libmill

Libmill is a relatively new library that introduces Go-style concurrency to C. It includes goroutines and channels. Consistent with Go, libmill's goroutines are cooperatively scheduled in userspace. However, whereas Go multiplexes goroutines across multiple threads as necessary, libmill's goroutines all run on a single operating system thread. Curious to compare its performance with other C implementations, we also implemented producer-consumer with libmill. We use libmill as a means of evaluating a producer-consumer model in C that utilizes userspace threads.[2]

6 BENCHMARKS

To benchmark our code, we first discuss the conciseness of producer-consumer implementations across languages, and the availability of simple threading APIs and thread-safe data structures. We then analyze the performance of producer-consumer implementations in each language.

6.1 Ease of Implementation

Though lines of code are typically an unreliable metric for assessing code, we still compare the lengths of producer-consumer implementations to highlight where preexisting threading and buffered queues support more concise code implementation. We count the number of lines necessary for each producer-consumer implementation by ignoring lines of code that are comments, prints, or command line parsing. We use the `cloc` tool, version 1.74 [1].

Given Go's preexisting concurrency tooling and buffered queues, an implementation of producer-consumer only required 55 lines of code.

The next shortest implementation was the producer-consumer model built with libmill in C, clocking in at 85 lines. This is unsurprising given libmill's implementations of goroutines and channels in C.

Our Rust implementation, which used a buffered queue from the `chan` crate but still required explicit thread spawning and joining, took 88 lines of code to implement.

All of these implementations were less than half of the length of the pthreads implementation, which took 175 lines of code to implement! Our pthreads C implementation was much longer given the lack of a buffered queue in the C library, meaning we had to implement this queue ourselves (adding more lines of code in the process). C also requires that programmers utilize mutexes and semaphores to ensure buffered queues are thread-safe. In comparison, Go, Rust, and the libmill library handle thread safety in channel implementations, so they are already safe for use in multithreaded programs. The availability of thread-safe data structures like buffered queues allows for more concise implementations in Go, Rust, and C with libmill.

6.2 Performance

To evaluate our code's performance, we benchmark it across two workloads that impact producer-consumer efficiency. The first simulates an I/O-bound workload, which relates to latency introduced by the time needed to handle file I/O or data reads. I/O latency affects producers as they read and preprocess data for consumers. This latency can leave the model's buffered queue empty, leaving consumer threads blocked and idle. The second simulates a compute-bound workload, in which the data processing done by consumers is computationally expensive and slows down consumers as they read from the buffered queue, therefore leaving the buffer full. Compute latency leaves producers blocked as they attempt to write data to an already full queue.

We benchmarked our code on an AWS EC2 instance with 2 virtual CPUs and 4 GB of RAM. All of our code was compiled with compiler optimizations: `gcc -Ofast` for C, `cargo build --release` for Rust, and `go build` for Go. We use the command line `time` tool to record total elapsed time, CPU seconds used by the producer-consumer process, and CPU seconds used by the operating system on behalf of the process. We used 5 producers, 5 consumers, and a buffered queue with a capacity of 100. We tested input sizes of 500, 5000, and 50000 items into each producer-consumer implementation.

6.2.1 I/O Bound. We modeled the I/O latency with a 1-millisecond `sleep()` call immediately before a producer adds an integer to the buffer. Our I/O bound tests (see Figure 1) show that, Go, Rust, and C's pthreads share comparable performance. Rust and pthreads use a substantial amount of system time to manage threading and the shared buffer. Conversely, libmill and Go do not even enter kernel space at any point–the operating system used 0 CPU seconds in every trial, instead spending more time in user space.

6.2.2 CPU/Compute Bound. We test compute bound by simulating latency when a consumer reads from the buffered queue and processes this data. We again introduce latency with sleeps of a 1 millisecond. These sleeps follow consumer reads. Consistent with the results from our I/O bound tests, Go, Rust, and C's pthreads shared comparable performance on the compute-bound benchmark (see Figure 2). Rust used the most amount of system time, followed by pthreads. As before, libmill and Go practically never enter kernel space, instead using user time to manage threading.

7 CONCLUSIONS

From a performance standpoint, our results do not indicate a particular "winner" among C (pthreads), Rust, and Go. The userspace-only libmill implementation was clearly the slowest due to its use of a single operating system thread. However, all of the other implementations were able to use multiple operating system threads. While we initially

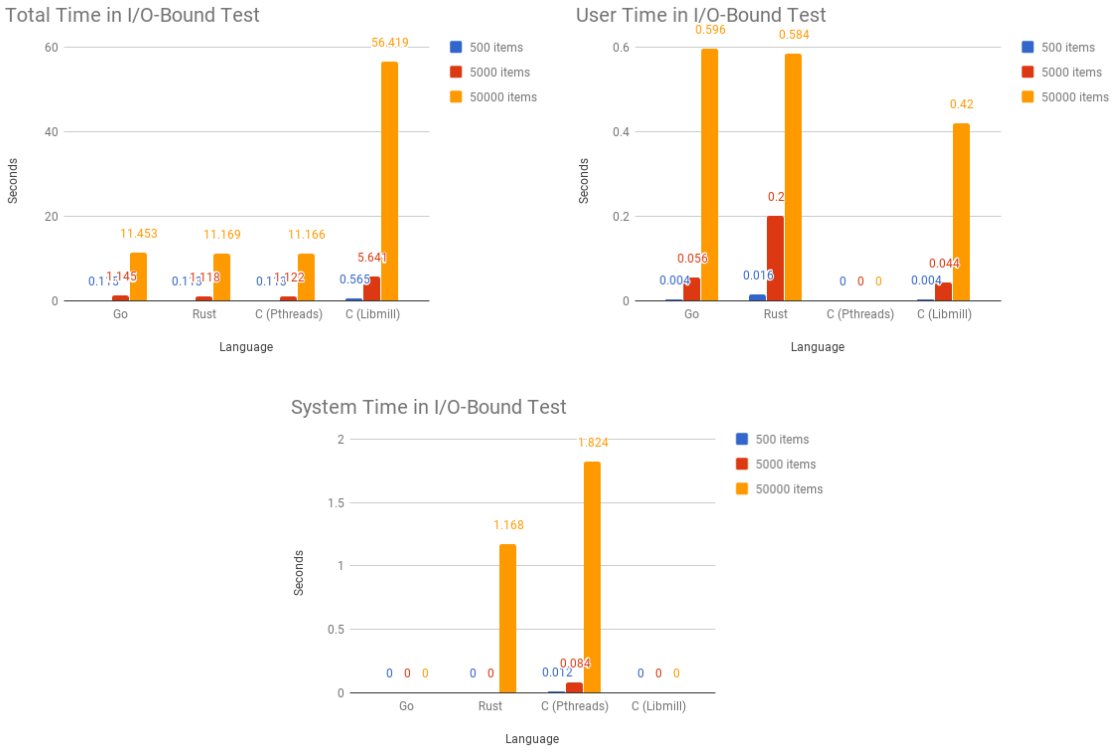

Fig. 1. For each producer-consumer implementation, we measured the total, user, and system time spent in a simulated I/O-bound workload.

hypothesized that Go's low-cost context switches between goroutines would give it a clear performance lead, our results suggest otherwise.

There are several reasons that we believe can explain the close results.

(1) Our performance benchmarks were small, contrived examples. Our benchmarks were intentionally designed to be relatively simple to implement, in order for us to have sufficient time to debug and test 4 implementations. As a result, they are not exactly representative of real-world use-cases of producer-consumer models. For example, a real-world use-case is less likely to simply add integers to a buffer and read them back out.
(2) We only ran our benchmarks with 5 producer threads, 5 consumer threads, and a buffer of size 100. Perhaps changing these parameters would show a greater difference in the threading implementations of each language.

Despite the lack of a single implementation with superior performance, we can conclusively agree that Go's native multi-threading constructs were the easiest to use for implementing our producer-consumer toy examples. Unlike in C and Rust, Go channels and goroutines are native to the Go language, do not require external libraries, and are extremely easy to setup. We did not have to worry about creating threads, memory allocation and deallocation across thread boundaries, and race conditions. These were real concerns that we faced in our C and Rust implementation, which often took us many hours to debug.

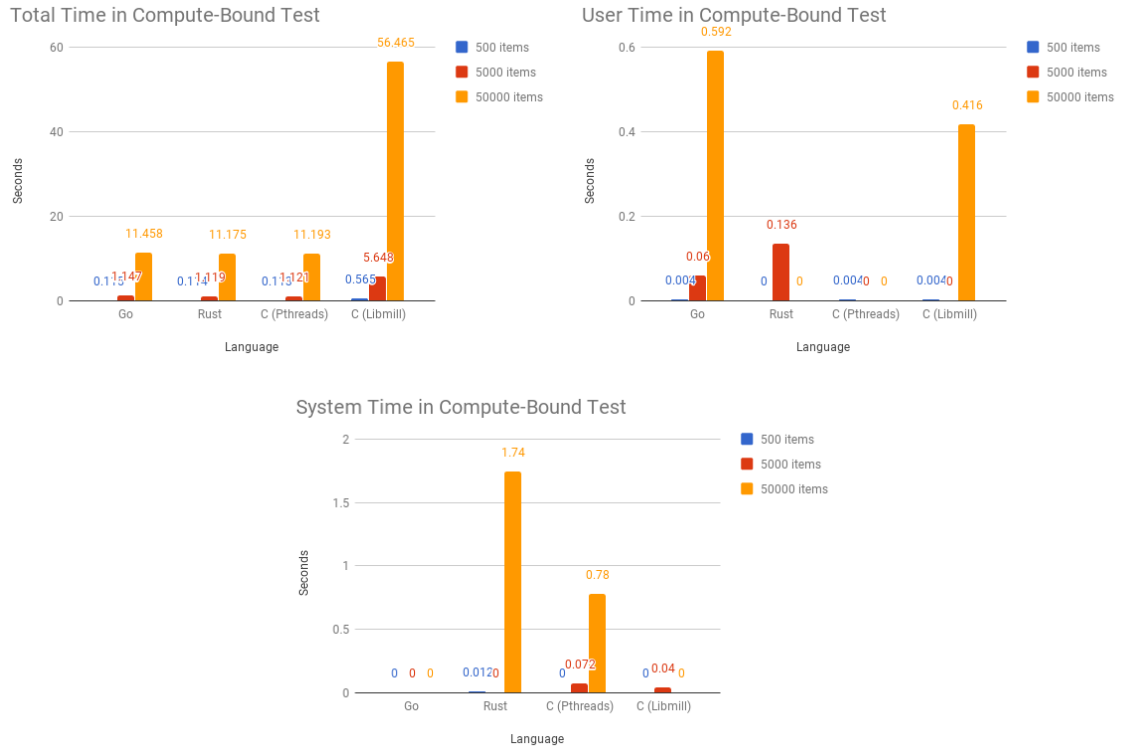

Fig. 2. For each producer-consumer implementation, we measured the total, user, and system time spent in a simulated compute-bound workload.

In conclusion, even without performance benefits, implementing producer-consumer models 4 times made us strongly believe in a core Go philosophy: "do not communicate by sharing memory; instead, share memory by communicating." [5] We hope that our exploration of threading constructs in the context of producer-consumer models may help future software developers evaluate different programming language options and paradigms.

ACKNOWLEDGMENTS

We would like to thank Will Crichton for teaching the CS 242 class, as well as our TA Jintian Liang for providing us with valuable advice and feedback.

REFERENCES

[1] [n. d.]. AlDanial/cloc. ([n. d.]). https://github.com/AlDanial/cloc
[2] [n. d.]. Libmill. ([n. d.]). http://libmill.org/
[3] Dave Cheney. 2017. Five things that make Go fast. (2017). https://dave.cheney.net/2014/06/07/five-things-that-make-go-fast
[4] Andrew Gallant. 2014. BurntSushi/chan. (2014). https://github.com/BurntSushi/chan
[5] Andrew Gerrand. 2010. Share Memory By Communicating. (2010). https://blog.golang.org/share-memory-by-communicating
[6] Krishna Sundarram. 2014. How Goroutines Work. (2014). https://blog.nindalf.com/posts/tech/how-goroutines-work/
[7] Dmitry Vorborev. 2016. Golang Channels Implementation. (2016). http://dmitryvorobev.blogspot.com/2016/08/golang-channels-implementation.html