Caitlin Scarberry
Audrey Dutcher

CS263B Writeup

For our project, we initially planned to benchmark how effectively popular open-source Go programs utilize Go's concurrency model. Over the course of the project, our goal shifted to creating benchmarks for Go's concurrency model itself. We aimed to analyze differences in the runtime versions and in the runtime implementations across platforms. To do this, we created a series of benchmarks that measured certain aspects of performance while the Go runtime was under heavy concurrent load. Our results have value to programmers, who could use them to determine the best compiler or platform to use for their program, and to the language developers, who could use them to measure the performance impact of changes to the runtime.

The Go language is a statically-typed, open-source language. Unlike many of the languages we looked at in class, it does not use a virtual machine to implement its runtime. Instead, the runtime is itself a Go library. This library implements features such as garbage collection and scheduling. Additionally, it has a public interface that Go programs can interact with. This enables Go programmers to, among other things, manually trigger garbage collection, yield to the scheduler, collect profiling information, and dynamically set the maximum number of logical processors that the program is allowed to use.

While the Go language does have some built-in benchmarks, none of them analyze the performance of concurrency. Proponents of the language often use Go's supposedly strong support for concurrency as a selling point; an aim of our project was to evaluate those claims.

For this project, our first step was to understand the Go runtime, particularly how it implements concurrency. Go's concurrency model is based on two primitives. The first primitive is a goroutine, which is effectively a lightweight thread that is multiplexed on top of operating system threads. Unlike threads in many other languages, goroutines have a flexible stack size, which is initialized at a low value. Thus, goroutines have a relatively low memory footprint, enabling millions of them to exist at once without causing a crash or significant slowdown, and can share a single core with much better efficiency than system threads. The second primitive is a channel, which is a way to asynchronously communicate between goroutines. It is essentially a blocking queue. If a goroutine attempts to send a value to a channel that is full, it will block until the channel has room for that value. If a goroutine attempts to read from an empty channel, it will block until that channel receives a value from another goroutine. Like goroutines, channels are extremely lightweight.

The Go scheduler is responsible for multiplexing goroutines onto operating system threads. It is work-stealing and non-preemptive. It uses three abstract data structures. None of these data structures are ever freed; instead, when no longer in use, they are put into a free pool of objects of that type. The first of these abstract data structures is a goroutine, the second is an operating system thread, and the third is a logical processor.

Each processor has a local queue of goroutines that are ready to run, and all processors share a global queue of other goroutines that are ready to run. When a processor needs work to do, it will try to get a goroutine from its local queue. If the local queue is empty, the processor will look in the global queue. Finally, if the global queue is empty, the processor will steal some goroutines from another processor's local queue. In order for a goroutine to execute, it needs to belong to a thread that has control of a logical processor. When a thread performs a system call, it yields the processor, and a new thread is created to utilize the processor so that the resource does not go unused. Once the original thread finishes its system call, it must wait to reacquire the processor.

With this knowledge in mind, we pivoted to constructing our benchmarks. Because part of our goal was to compare performance across machines and platforms, we preferred relative benchmarks over absolute benchmarks. Additionally, we observed that good benchmarks are ones which reflect how the language is used in practice.

Our first step in creating the benchmarks was to determine the common use cases for Go concurrency. To do this, we analyzed several popular open-source Go programs. The programs we looked at included Go's own built-in HTTP package, several web frameworks, a text editor, and a SQL database. The most popular uses of concurrency that we found were: to perform CPU-intensive tasks in parallel, to asynchronously move data between goroutines using a pipeline of channels, and to perform an action after a timeout. Using this list of common use cases, we determined that we wanted to measure four properties of the runtime: the parallel efficiency of the scheduler when handling large numbers of concurrent and independent goroutines; the time for a value to be passed through each channel in a data pipeline; the accuracy of timeouts when the scheduler is under a heavy load of other goroutines; and the cost of starting a goroutine relative to the cost of communicating through a channel.

We then began to implement our benchmarks in Go. At first, we utilized the benchmark driver used in existing Go benchmarks. However, it soon became apparent that this driver was not entirely suitable for our purpose, and we ultimately decided against using it. We made our benchmark parameters fully configurable from command-line flags in order to more easily look for interesting results. Additionally, each benchmark program monitored and recorded its CPU and memory usage statistics. In total, these benchmarks were over four-hundred lines of code.

Once we had our benchmarks, we needed to determine our testing environment. This environment was a matrix of operating system, compiler, and runtime version. We considered using CSIL as our Linux environment, but in the end, we decided to only use our personal laptops: one MacBook Pro with a four-core 2.7 GHz Intel Core i5 and one Dell with an eight-core 2.9 GHz Intel Core i7. The Dell ran both Linux and Windows. We compiled our benchmarks using both the official gc compiler, which was created by the creators of Go, and the gccgo compiler, which was created by the makers of GCC and is only available for Linux. For gc, we compiled our code with versions 1.4 through 1.10, which correspond to the official versions of the Go runtime. For gccgo, we compiled our code with versions 5 through 8, none of which corresponded exactly with an official version of the Go runtime. In total, this matrix gave us twenty-five sets of data for each benchmark.

Our parallel efficiency benchmark launched a large number of independent goroutines executing a simple for-loop, timed how long they took to complete on one processor, and then compared that to how long they took to complete on an increasing number of processors. We found that code compiled by gc had a sharp decrease in parallel efficiency after half of the machine's cores were utilized, both on the four-core and eight-core machine. We also found that gccgo did significantly better than gc; the lowest parallel efficiency of gccgo was .7, while the lowest parallel efficiency for gc on the same machine was .4.

Our data pipeline benchmark called a recursive function which set up a pipeline of goroutines, each of which took a value from an input channel, performed a simple arithmetic operation, and then transmitted the result to the next goroutine's input channel. We timed how long each step of the pipeline took as the number of steps in the pipeline increased. Across all operating systems, versions, and compilers, the time per step was largely unaffected by the number of steps. We had two outlying results. On Windows, the time per step for gc version 1.4 was more than ten times the time per step on all subsequent versions. This indicates that a significant improvement was made to the Windows implementation of the compiler or runtime between 1.4 and 1.5. Additionally, channel use in gccgo took approximately three times longer than the average channel use in gc.

This result illustrates how these benchmarks may be useful to programmers: a program that relies heavily on parallel efficiency would likely perform better if compiled in gccgo, but a program that relies heavily on channel usage would likely perform better if compiled in gc.

Our timeout benchmark launched a goroutine with a timeout, launched a large number of other goroutines, and then compared the time at which the timeout should have been triggered versus the time it was actually triggered. With 1000 goroutines, there was no significant delay; however, with 16,000 goroutines, there was a slowdown of over 1000 percent. This is likely due to the Go scheduler being non-preemptive. We verified this by manually inserting calls to the scheduler into the CPU-intensive goroutines, which caused the timeouts to fire at the correct time regardless of load. This result contains useful information for Go programmers: timeouts cannot be relied upon to trigger at the exact moment they are set to.

Our benchmark for comparing the cost of starting goroutines to the cost of channel communication was more complex than our other benchmarks. It measured the difference between runtime when a mathematical task was completed by initiating a new goroutine for each step versus when the same task was completed by having several persistent goroutines which acted as workers and were communicated with through channels. For gc 1.5 and 1.4, it was significantly more efficient to initiate new goroutines than to communicate with persistent ones through channels. However, in all later versions of gc, there was no significant difference. Thus, in modern Go, both concurrency primitives -- goroutines and channels -- are equally efficient. This is an indication that Go does indeed have the strong support for efficient concurrency that it claims to have.

Through these benchmarks, we were able to learn information about Go that could be of use to Go programmers and the developers of the language itself. We concluded that Go’s claims of having efficient concurrency are accurate, and that the built-in concurrency primitives are the only tools necessary for creating efficient concurrent programs in Go.