CSC564 Concurrency (Fall 2018)

Assignment 1

Prof. Yvonne Coady

Spencer Rose (V00124060)
October 11, 2018

Problems in Concurrency

Introduction

Analysis Overview

Correctness[i]

1. **Safety**: Code demonstrates the avoidance of deadlocks, livelocks and runtime errors;
2. **Liveness**: An action is eventually executed that follows fair choice and action priority (avoidance of starvation).
    - **Fairness**: Choice over a set of transitions is executed infinitely often, then every transition in the set will be executed infinitely often.

Comprehensibility

1. **Code Readability**:
    - Lines of Code (LoC): Though not a well-liked metric, LoC gives a rough idea of the overall complexity of a system.
    - Numbers of synchronization primitives and channels offer indications of difficulty, and effort of a piece of code; flow complexity; lines of code. (NOTE: n is the number of goroutines/threads using a primitive concurrently)
2. **Maintainability**:
    - Code shows coupling/cohesion: smaller modular units and demonstrates ease of testing;
    - Code is amenable to error detection

Performance

1. **Running Time:** This is the total CPU time (i.e. User + System time) measured at completion of the process. Note that since these tests were run on a multi-core processor, elapsed time was not a reliable indication of running time.
2. **Heap Allocation:** Total heap allocation at runtime (indicates a memory "footprint").
3. **Waiting time:** Wait time is the time threads or processes spend waiting for another thread to perform an action. Performance issues in multithreaded programs such as competition for resources, synchronization and scheduling problems, increase the overall waiting time.[ii] Though I did carry out an analysis of C-based implementations using mutrace (see below), I did not have enough time to complete an analysis of Golang wait times using pprof, so this (partial) analysis is omitted.
4. **Profilers:**
    - pprof [Golang] (https://golang.org/pkg/net/http/pprof/) to provide runtime profiling data (CPU usage, memory allocation, mutex contention)
    - mutrace [C] (http://0pointer.de/blog/projects/mutrace.html) to monitor runtime and mutex contention
    - valgrind [C] to monitor memory allocation.

Specifications

Processor	MacBook Pro (15-inch, 2018) 2.2 GHz Intel Core i7
Operating System	**Problems 1-5:** Linux precise64 3.2.0-23-generic #36-Ubuntu SMP Tue Apr 10 20:39:51 UTC 2012 x86_64 x86_64 x86_64 GNU/Linux **Problem 6:** Darwin Kernel Version 17.7.0: Fri Jul 6 19:54:51 PDT 2018; root:xnu-4570.71.3~2/RELEASE_X86_64 x86_64
GNU Compiler	gcc version 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)
Golang	go version go1.11 darwin/amd64
Python	Python 3.6.5 64bits, Qt 5.9.4, PyQt5 5.9.2 on Darwin

PROBLEMS

Generalized Cigarette Smokers Problem (Exercise 4.5.4)

The original Smokers Problem was conceived by Suhas Patil[iii] (1971) to illustrate a multi-process synchronization problem that could not be solved using semaphores (i.e. without the use of conditional statements). This is an impractical restriction, according to David Parnas[iv]. However, the problem highlights the value of the semaphore's "nondeterminism," in that when there are multiple threads waiting on a semaphore, the "V" operation does not specify the process to be released. This is useful for schedulers subject to uncertainty and change, or applications that may be removed from a scheduling queue, while maintaining smooth (deadlock-free) operation of resource production and consumption. Downey likens the problem to how operating systems allocate resources to applications: the agent representing an OS (scheduler) that allocates resources to running applications (smokers). Synchronization between the agent and applications (smokers), in this example, requires that waiting applications only proceed when the correct resources are available. At the same time, we wish to avoid waking applications that cannot proceed, demonstrating simultaneous wait. The solution proposed by David Parnas: to create "helper" or "pusher" threads that wake waiting smokers, acts as a communication bridge between the agent and smoker. In the generalized version of the problem (implemented for the assignment), the agent does not wait for a response (uptake) when resources are made available.

Implementation A (Golang)

- https://github.com/scrose/csc564/blob/master/a1/cigarette_smokers_problem_A/cigarette_smokers_problem_A.go
- Adapted from the solution in Downey, p.101.
- Uses six channels (two for each ingredient) to synchronize resources.
- The agent only signals available resources along helper channels (one for each ingredient) that keep track track of available ingredients using a scoreboard.
- Each smoker communicates with a designated helper through an independent channel.
- The use of the **select** statement allows simultaneous selection of signals, which is not available in C.

Implementation B (C)

- https://github.com/scrose/csc564/tree/master/a1/cigarette_smokers_problem_B
- Similar adaptation from the solution in Downey, p.101, but using semaphores in a way analogous to channels.
- Each smoker communicates with the designated helper through an independent channel.
- An additional "ready" semaphore is added to let the agent know that a resource was used.

Analysis: Generalized Smokers Problem

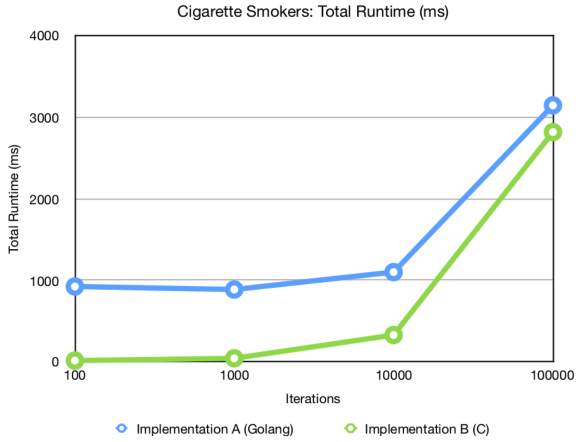
Correctness

	Implementation A (Golang)	Implementation B (C)
Safety	The use of intermediary "helper" goroutines avoids deadlock between smokers needing overlapping ingredients. Only shared data is a scoreboard locked using two mutexes (used separately) to: (1) update the ingredient count; (2) update the number of completed smokes. Only helpers update this shared data. The agent freely release ingredients without communication with helpers or smokers.	The use of intermediary "helper" threads avoids deadlock between smokers needing overlapping ingredients. Only shared data is the ingredients scoreboard locked using two mutexes (used separately) to: (1) helpers update the ingredient count; (2) smokers update the number of completed smokes. The agent releases ingredients anytime it receives a "ready" signal from the helpers.
Liveness	The resources released by the agent are randomized to ensure an equal spread is available, while the helper threads have equal access to these resources.	The resources released by the agent are randomized to ensure an equal spread is available, while the helper threads have equal access to these resources.

Comprehensibility

LoC / Func.	225 /	466
Complexity	7 goroutines (1 agent, 3 helpers, 3 smokers) 6 channels 1 mutex	7 threads (1 agent, 3 helpers, 3 smokers) 6 semaphores 2 mutexes
Overview	Since Golang channels and mutexes do not require extensive initialization, the code is cleaner and more compact when compared with the C implementation. Functions are less verbose and	Initialization of synchronization primitives is verbose in C, though more encapsulation is possible with this implementation.

Performance

Runtime	Implementation B (C) performed faster on all tests for different iteration counts – in this case, the number of completed (smoked) cigarettes. Since a fixed number of threads/goroutines (7), the runtime allocation for heap was also relatively constant across iteration counts. Interestingly, the heap allocation was considerably higher with the Golang implementation, yet heap cost depends on the object layout of the runtime system independent of the processor.	
Memory	Heap Allocation (stable): ~296kB	Heap Allocation (stable): ~4.4kB

FIFO Barbershop Problem (Exercise 5.3)

The Barbershop problem is a classic multi-process scheduling problem that emerges in operating systems. The problem specifically models difficulties in synchronizing actions (without deadlock) that have an unknown amount of time. For example, an arriving thread or process may be about to be queued by the scheduler while an OS is running an application. In the process of being queued, however, the running thread may end that results in an empty queue. The scheduler therefore finds no waiting threads and goes to sleep. The scheduler and the thread are both waiting, i.e. deadlock. A less likely example is when two threads are vying to be queued at the same time when there happens to be only space or "seat" in the waiting room. Furthermore, we may know, or be able to specify, the OS wakeup policy of waiting threads (e.g. some implementations of futex), and so we cannot assume the order that queued threads are released. Using a application FIFO queue, as illustrated in the following implementations, resolves this potential by ensuring the correct order of queued threads.

Implementation A (Golang)

- https://github.com/scrose/csc564/tree/master/a1/fifo_barbershop_A
- A buffered channel (channel-in-channel) is used as a FIFO queue of bounded capacity (number of seats in the waiting room).
- Channels are employed for the barber and customer to wait and signal the start and end of a haircut.

Implementation B (C)

- https://github.com/scrose/csc564/tree/master/a1/fifo_barbershop_B
- Adapted from Downey, p.127.
- Required a FIFO queue implementation locked by a mutex.
- The customers enqueue themselves (unless the queue is full, and therefore exit); the barber pops customers off the queue.
- Each customer has a semaphore that is passed to the queue and used by the barber to signal each customer separately.
- Once the haircut is finished, the barber and customer interleave a receive-signal-confirmation pattern that ensures the threads proceed in lockstep (the customer exits, and the barber proceeds to the next customer in the queue).

Analysis: FIFO Barbershop Problem

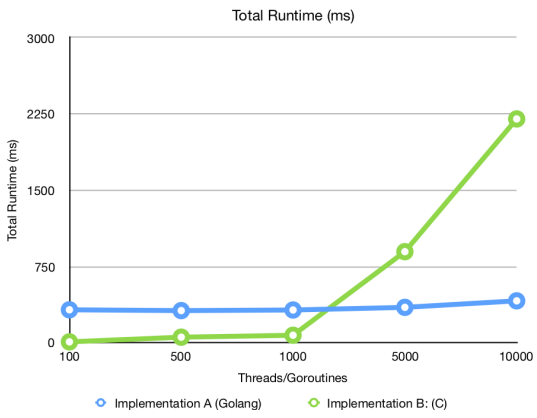
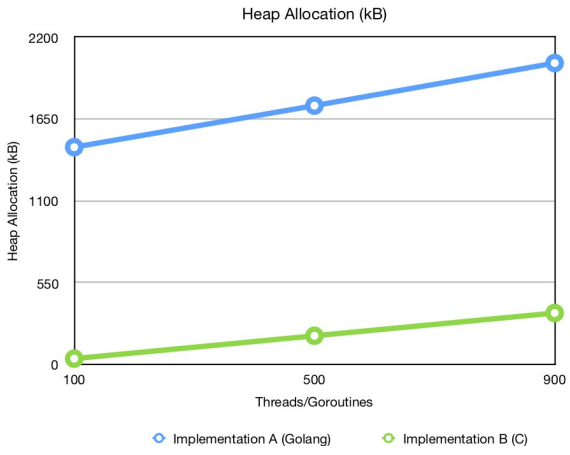
Correctness

	Implementation A (Golang)	Implementation B (C)
Safety	Go's buffered channels simplify this problem immensely, compared with the C version. Buffered channels guarantee a fixed-size FIFO queue to limit concurrent usage, and which can be controlled (wait-signal) as a channel. This ensures the barber does not pop an empty queue, and enqueuing customers do not encounter full queues. It also ensures we do not leave the order of the queue to the OS scheduler (see implementation notes above).	When customers enqueue, they add a semaphore to the queue used by the barber to communicate with the customer (similar to a two-thread rendez-vous pattern). Each customer thread therefore waits on its own semaphore. This ensures that only the correct next customer is signalled by the barber. The individualized semaphores are analogous to channels-in-channels (channels sent over channels in Go).
Liveness	As required by the problem, customers are served in the order they arrive (by the buffered channel), and therefore no starvation will occur. However, when customers are arriving at too rapid a rate, most balk.	As required by the problem, customers are served in the order they arrive, and therefore no starvation will occur, so long as the barber progresses.

Comprehensibility

LoC	108	332 + tracker
Complexity	n + 1 goroutines / 2 channels (buffered, channel-in-channel) / 2 mutexes	n = 1 threads / 3 + n (customer nodes) semaphores / 1 mutex
Overview	The buffered channel also greatly improves the readability of the code by removing the need for a queue implementation.	Semaphore and mutex initialization, as well as the queue implementation details, contributes to long and complex code.

Performance

Runtime/ Memory		
Overview	Implementation A runtime remained surprisingly low and constant over the number of goroutines generated. Implementation B reached a threshold	Implementation A heap allocation was roughly 1.5kB more than Implementation C for three test samples. This suggests Go has greater overhead in its API over

	around 1000 threads and the runtime increased more than the factor of thread increase.	the leaner C runtime.

Building H_20 (Exercise 5.6)

The H_2O problem focuses on the synchronization pattern of the barrier. Barriers block threads until a threshold number of threads are waiting, at which point all waiting threads are released. Barriers are useful when you want a number of tasks to be completed before an overall phase of work can proceed. For example, parallelized calculations require local sequential calculations to be computed before a parallel calculation can execute (see Naive Bayes Classifier below). Similarly, the separate phases of a multi-phase algorithm might require the coordination and barrier synchronization of independently running sub-tasks. A barrier can force all of the threads that are doing parallel computations to wait until all involved threads have reached the barrier. When the threads have reached the barrier, the threads are released and begin computing together. In the following implementations, we can see that a simple wait call is insufficient to match threads in a structured triplet (H_2O) that is released once all of the correct component threads are assembled.

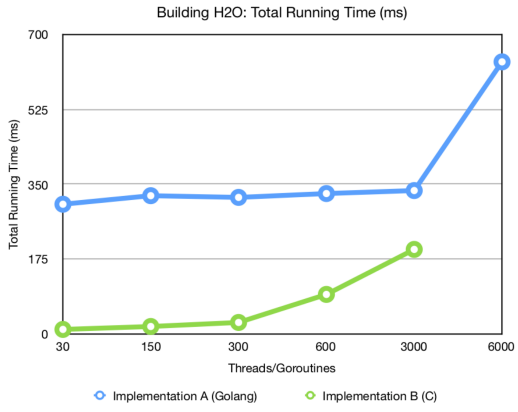
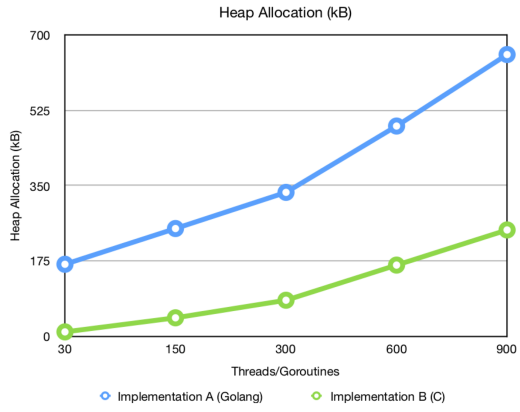
Implementation A (Go)

- https://github.com/scrose/csc564/tree/master/a1/building_H2O_A
- For each goroutine “bond”, a channel is sent within a channel to facilitate one-to-one communication between the three entities.
- A simplified barrier that uses channels is employed to force the aggregating goroutines to wait for the correct adjoining goroutines.
- The "oxygen" goroutine coordinates the bond of two "hydrogen".
- Barrier is released once the bonding channel is closed by the last goroutine to join.
- A scoreboard locked by a mutex tracks the number of atoms and formed H_2O molecules.

Implementation B (C)

- https://github.com/scrose/csc564/tree/master/a1/building_H2O_B
- Adapted from Downey, p.148.
- Uses a pthread barrier to synchronize the assembly of triplet threads.
- A scoreboard locked by a mutex tracks the number of atoms and formed H_2O molecules.

Analysis: Building H_20

	Implementation A (Golang)	Implementation B (C)
Correctness		
Safety	Employs a barrier pattern using a channel as the waiting mechanism. The last thread to reach the barrier closes the channel which effectively broadcasts the release of all waiting threads. Each oxygen creates a barrier that is sent over two hydrogen channels. This ensures the three goroutines share the same barrier, and obviates the need for a reusable barrier. This implementation also removes the variability of which goroutine unlocks the barrier mutex.	POSIX pthreads specify a (reusable) barrier synchronization object with functions to create the barrier for a specified number of threads. When the last thread arrives at the barrier, all the threads resume execution. Note that (Downey, p.148) the unlocking of the mutex is variable, since threads may lock it, update the counter, and unlock it. Once a thread arrives to form a complete set, it has to lock the mutex to block subsequent threads until the barrier is released. However we are guaranteed that the oxygen will release the mutex.
Liveness	There is no requirement of an order to the "bonding" of goroutines. A locked scoreboard ensures all available hydrogens and oxygens are visible.	There is no requirement of an order to the "bonding" of goroutines. A locked scoreboard ensures all available hydrogens and oxygens are visible.
Comprehensibility		
LoC	161	211
Complexity	3n[1] goroutines / 3n channels (ch-in-ch)	3n threads / 2 semaphores / 1 pthread barrier
Overview	The use of channels simplifies how we understand the communication between the assembling goroutines, e.g. we can specify exactly two channels to correspond to two hydrogen molecules.	The pthread barrier object simplifies the implementation, though it is only available on certain POSIX implementations.
Performance		
Runtime/ Memory		
Overview	Implementation A is slower but can handle 6000	Go's overhead requires greater memory usage than

1 For n goroutines.

	concurrent entities, whereas C chokes at 3000 concurrent threads.	the C implementation.

Search-Insert-Delete Problem (Exercise 6.1)

The Search-Insert-Delete problem is a variation of the readers-writers problem using multiple categorical mutual exclusion: mutexes determine which category of thread allowed in the critical section at any time. As Downey states, “A thread in the critical section does not necessarily exclude other threads, but the presence of one category in the critical section excludes other categories.” Hence, the problem demonstrates different allowable types of concurrent access by the different types of computational entities. The problem is illustrative of any situation in which multiple processes try to access shared data, such as a data structure or database, where we want to ensure the reading of the data is consistent and valid. The Search-Insert-Delete problem includes an additional category: the Inserter, which runs concurrently with searchers (essentially readers), but excludes deleters (writers) and other inserters. As categorical exclusion problems are prone to asymmetric solutions, in which one category of thread blocks others from progressing, starvation is probable. Specifically, deleters must wait for searchers and inserters - which can run concurrently - to complete before locking the linked list. This problem can be eased by prioritizing one category (e.g. writers), or, as presented in Implementation B, by combining concurrent design patterns such as multiplexes, which limit the number of concurrent processes in the critical area, and turnstiles, which allow one thread in at a time. It's not clear whether an optimized solution to fairness is possible.

Implementation A (Golang)

- https://github.com/scrose/csc564/tree/master/a1/search_insert_delete_A
- Uses Go's RWMutex, which ensures a blocked Lock call excludes new readers from acquiring the lock.
- If a goroutine holds a RWMutex for reading and another goroutine might call Lock, no goroutine should expect to be able to acquire a read lock until the initial read lock is released to ensure that the lock eventually becomes available. This has some effect on avoiding the starvation of deleters: if a searcher is in the critical section, it forces deleters to queue, but not other searchers. So, if a deleter arrives it can lock searchers, which will cause subsequent searchers to queue.

Implementation B (C)

- https://github.com/scrose/csc564/tree/master/a1/search_insert_delete_B
- Uses intersecting concurrency patterns of Lightswitch, a synchronization technique that employs a First-In-Last-Out principle for threads onto a semaphore, so that group of threads can collectively request access to the critical section, and release it when the group has finished processing.
- Required a linkedlist implementation.

Analysis: Search-Insert-Delete

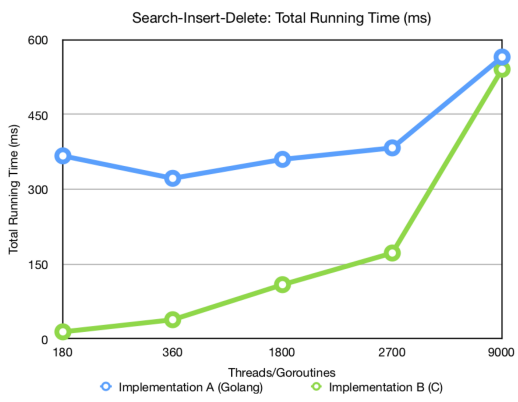
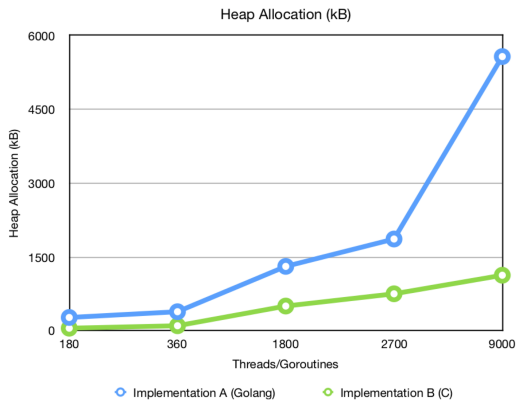
Correctness

	Implementation A (Golang)	Implementation B (C)
Safety	Uses Go's RW mutex read lock for searchers. If a goroutine holds a RWMutex for reading and another goroutine might call Lock, no goroutine should expect to be able to acquire a read lock until the initial read lock is released.[v] Inserters use a (regular) mutex to exclude each other, and the searcher read lock. Deleters use the write lock on the RWMutex, which excludes it from both inserters and searchers.	This implementation works using the synchronization pattern of FILO lightswitch (see definition above) to effect a read lock on searchers. Inserters require both a searcher lightswitch and an inserter mutex to limit inserters to one, but allow readers. Deleters only require an inserter mutex to block all. Thus, searchers retain access, but deleters and other inserters are excluded. The lightswitch for searchers ensures FILO concurrent access. Deleters wait on the "empty" attribute (i.e. No searchers or inserters) held by the searcher lightswitch.
Liveness	As an experiment, a bounded bypass is added to prevent deleter starvation: a max number of searchers can be granted access to the linkedlist at one time once a deleter has enqueued.	Since a steady stream of searchers can enter the critical section, deleters require extensive wait time before the lock is released. This can be mitigated with a bounded turnstile that bars additional searchers once a deleter has queued.

Comprehensibility

LoC	187	365 + tracker[2]
Complexity	3n goroutines / 1 ch / 3 mutexes (RW mutex)	3n threads / 1 semaphore / 3 mutexes
Overview	This implementation is extremely short and easy to follow as it uses Go's builtin RWMutex and List packages. The complexity comes out of the simulation of random events, such as a randomized search of the list.	Concurrency patterns offer a systematic approach to forming solutions that make it easy to follow. Once the pattern is encapsulated in an object, it is easy to use and understand the algorithm flow.

Performance

Memory		

2 This is a small set of minimally-invasive timing functions that test wait times of various parts the program.

Overview	Imp. B out-performs A for < 9000 entities.	Memory allocation is higher for Golang as we've seen with the overhead. May be due to **defer** overhead.[vi]

Baboons Crossing (Exercise 6.3)

Given two types of threads A and B, this problem models how a single resource can only be shared concurrently (i.e. accessed as read-only) by a fixed number of threads, and furthermore, one type at a time. Threads of one type, say type B, are locked out from accessing the resource until no threads of type A are in the critical section. This problem might model a two-way serial communication channel that is exchanging data between servers. As with the Search-Insert-Delete problem, starvation of one type of thread or process is a significant possibility, if, for example, a stream of type A threads locks the resource indefinitely. This is equivalent to only one side of a communication channel being able to efficiently transmit data. To mitigate the problem, we want to ensure a fair interleaving of both types of threads (baboons going East and West get equal rope time). In Implementation B, a synchronization scheme that combines design patterns turnstile, lightswitch and multiplex can facilitate fairness by ensuring one type does not dominate access to the resource.

Implementation A (Golang)

- https://github.com/scrose/csc564/tree/master/a1/baboon_crossing_A
- Mutex-free implementation that uses Go's **select** functionality to wait on multiple channels simultaneously, which I'm not aware of an analogue to that in C.
- A "rope" goroutine is both the shared resource and acts as a monitor by tracking waitlists on both sides and the number of crossing baboons at one time, while granting the fixed number of concurrent baboons.
- The baboon goroutine is quite minimal – it merely requests access to the rope, waits for a response, and confirms that it has crossed, all on one channel. The tradeoff is that the rope goroutine is long and complicated channel selection. (Note, however (as outlined in the Golang documentation[3]), if multiple requests are waiting, one of them is chosen by the **select** statement at random (pseudo-random).

Implementation B (C)

- https://github.com/scrose/csc564/tree/master/a1/baboon_crossing_B
- Uses intersecting concurrency patterns:
  - Lightswitch: Employs a First-In-Last-Out principle for threads onto a semaphore, so that group of threads can collectively request access to the critical section, and release it when the group has finished processing.
  - Turnstile: Semaphore wait-signal that allows only one thread to proceed at a time.
  - Multiplex: A pre-loaded semaphore that allows X number of threads in the critical section at one time.
- Inverse of Implementation A, since the baboon threads do all the work, and the rope thread is only used to (optionally) display scoreboard information at specific intervals.

3 See: https://golang.org/ref/spec#Select_statements

Analysis: Baboons Crossing

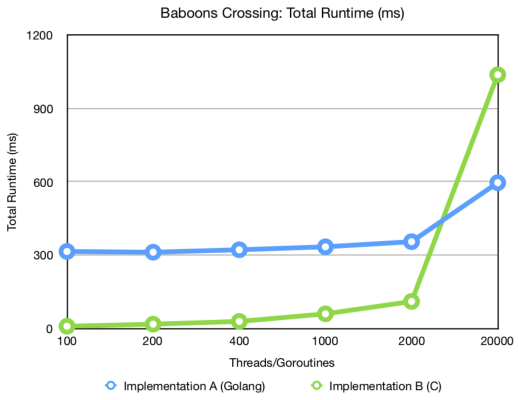
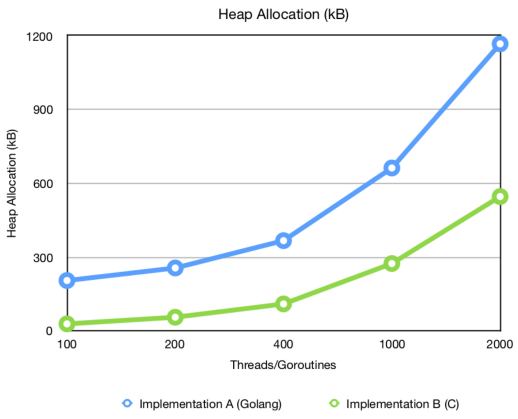
Correctness

	Implementation A (Golang)	Implementation B (C)
Safety	Using a single select statement, the main "rope" routine can act as a monitor to control access to the rope, by signalling the appropriate waiting baboon goroutines. By centralizing the control, no locks are needed for the scoreboard (tracking crossing baboons, direction, etc.). The baboon goroutines are minimal, only dispatched by the rope.	Each direction has a lightswitch (see above) that makes the rope available only to concurrent baboons going that direction; a multiplex (semaphore) limits that number to five. As noted by Downey (p. 3) concurrency patterns offer a systematic approach to assembling solutions that are demonstrably correct. As mentioned, the baboon threads synchronized between themselves.
Liveness	To prevent starvation of one direction, a rope lock flag is used to stop the number of crossing baboons at 5 when there are goroutines waiting to travel in the other direction. Note that the order of the crossing baboons was not a requirement.	A turnstile wraps each lightswitch acting as a"bounded bypass": if the lightswitch is locked, blocked threads will get "stuck" in the turnstile, forcing newcomers to queue, such that blocked threads will eventually gain access. Thread tracking showed the ratio of East to West thread waiting time was 0.9807.

Comprehensibility

LoC	181	324
Complexity	5 channels (channels-in-channels)	2 lightswitches / 1 turnstile / 1 multiplex /
	As mentioned above, the main channel select statement is long, but essentially maps the nondeterminism of the channel communication into a tree that can be read sequentially.	A feature of concurrency is their readability, since the complexity of the semaphore signalling is encapsulated. Patterns make this implementation clearer to understand. Likewise, the two baboon thread routines are mirror images, reducing the complexity.

Performance

Runtime/ Memory		
Overview	Implementation A outperformed B except at higher thread counts. Hopefully this pattern shows the limits of C multi-threading, but also	Implementation A held a margin of additional heap memory over B, with a steeper increase at higher goroutine counts.

	the overhead processing of Golang's runtime.	

Naive Bayes Classifier

This problem could be classified under the Single Program Multiple Data (SPMD) model: the parallelization of an algorithm by dividing up tasks and running them simultaneously on multiple processors. The goal is to achieve faster results for the computation over a serial implementation of the algorithm. In the following analysis, two bottlenecks were identified in a serial implementation of the Naive Bayes text classification algorithm using a timing probe. Naive Bayes is a common text classification method used in text data mining to categorize documents into classes based on an analysis of the vocabulary's conditional probabilities for given classes. It has two phases: (1) a learning phase that builds a probability model using training documents; (2) An application phase that applies the model to classify test documents.

As text classification can often involve the processing of massive datasets comprised of thousands or millions of documents, performance is a critical factor in a successful implementation. Though not all classification algorithms can be parallelized (for example, the Perceptron is resistant to parallelization), researchers have developed concurrent algorithms for Naive Bayes.[vii] The approach in the following was to identify non-recursive iterations that can split up into tasks handled by a multiple threads (i.e. worker pool pattern).

Following a timing analysis of for-loops in the sequential implementation, two significant bottlenecks were identified in the data training and model application sections of the algorithm. I decided to investigate the differences between using Python's multiprocessing and multithreading packages to determine if any improvements to performance were achievable. This NB Classifier used in this example was adapted from a serial implementation posted online.[viii] The solution was tested on a Macbook laptop equipped with a Intel "Core i7" processor, with six independent processor "cores".

Bottlenecks:

1. NB Training routine: Eliminating the “stop words” from set of documents vocabulary
2. NB Application: Calculating the prediction probability scores for each document.

Implementation A

- Uses Python 3.6's multiprocessing package.
- https://github.com/scrose/csc564/tree/master/a1/naive_bayes_A

Implementation B

- Uses Python 3.6 's multithreading package
- https://github.com/scrose/csc564/tree/master/a1/naive_bayes_B

Analysis: Naive Bayes Classifier

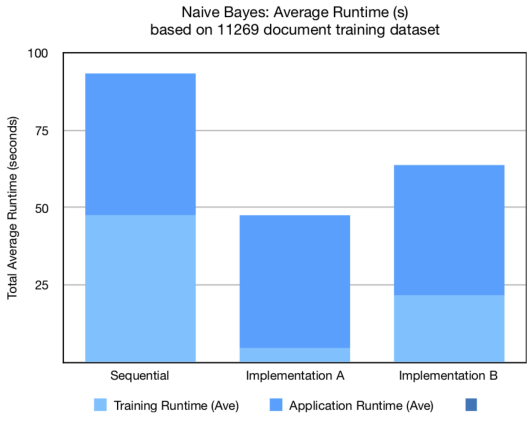
Correctness

	Implementation A (Python)	Implementation B (Python)
Safety	Correctness of the Naive Bayes Algorithm is not the focus of this analysis; however, the classification results were verified on Weka data mining software. The results of Implementation A matched the results of the sequential implementation.	Correctness of the Naive Bayes Algorithm is not the focus of this analysis; however, the classification results were verified on Weka data mining software. The results of Implementation B matched the results of the sequential implementation.
Liveness	Not applicable.	Not applicable

Comprehensibility

LoC	~300	~300
Complexity	Processes: 21 High complexity: uses NumPy and Pandas packages for swift matrix manipulations and calculations. Implementation A follows the same calculations as the sequential version.	Threads: 21 High complexity: uses NumPy and Pandas packages for swift matrix manipulations and calculations. Implementation A follows the same calculations as the sequential version.

Performance

Runtime/ Memory		Multiprocessing improved the average overall running time substantially with multiprocessing and less so with multithreading. A possible explanation is that processes (forking) speed up Python operations that are CPU intensive because they benefit from multiple cores and circumvent the Global Interpreter Lock (GIL). In other words, Python's GIL prevents performance gains for CPU-intensive executions, which is what hampered the performance gains of multithreading in Implementation B. Note that NumPy and Pandas could not be further optimized, resulting in no improvements to Bottleneck 2. Based on 10 trials. The sample dataset size was 11269 training documents.

References

i Lamport, Leslie, Verification and Specification of Concurrent Programs,, 16 November 1993, accessed on September 24, 2018, https://lamport.azurewebsites.net/pubs/lamport-verification.pdf.

ii Ji, M., Felten, E. W., & Li, K. (1998, June). Performance measurements for multithreaded programs. In *ACM SIGMETRICS Performance Evaluation Review* (Vol. 26, No. 1, pp. 161-170). ACM.

iii Patil, Suhas S. (February 1971). Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination among Processes (Technical report). MIT, Project MAC, Computation Structures Group. Memo 57.

iv Parnas, David L. (March 1975). "On a solution to the cigarette smokers' problem (without conditional statements)" (PDF). Communications of the ACM. 18 (3): 181–183.

v Golang specifications, https://golang.org/pkg/sync/#RWMutex.Lock.

vi See: https://medium.com/i0exception/runtime-overhead-of-using-defer-in-go-7140d5c40e32, retrieved September 29, 2018.

vii For example: Kruengkrai, Canasai and Jaruskulchai, Chuleerat, A Parallel Learning Algorithm for Text Classification, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.13.3374&rep=rep1&type=pdf

viii Nazrul, Syed Sadat (retrieved Sept 20, 2018) https://towardsdatascience.com/multinomial-naive-bayes-classifier-for-text-analysis-python-8dd6825ece67