Matthew Paletta Oct 11, 2018

Assignment 1

All of the code for this assignment, and the testing framework described here are available at: https://github.com/mattpaletta/Little-Book-Of-Semaphores

For writing my Go code, and inspecting code I found online, I used GoLand, a Jetbrains IDE for Go. Likewise, for Python, I used PyCharm Professional.

For my testing, I tried to automate it as much as possible, so as to allow for replication by another user later. My testing framework is also available on Github.

The testing framework I wrote is written in Python. All of the code for it is located in testing.py, available in the repository. The requirements file is also available there for replication. First I'd like to discuss what the testing framework does, then some of the inspiration, or rationale behind certain choices I made while designing it.

The testing framework first looks in the current directory ("problems"), and picks out any tests, specified by folders. For every test, it iterates over the files in that directory, picks out the language for that file based on the file extension, and generates a docker image to run for that test.

The docker image is version locked as much as possible, so it's replicable. If the test is written in Go, it uses golang:1.11.1-alpine. If it's written in Python, it uses both python:3.6.6-slim-jessie and pypy:3-6.0.0-slim-jessie. The jessie and alpine images are just the most trimmed-down versions of the base images. So they save space when you download them, and don't take as long to start up, which is very important. The benchmark, or baseline image I'll explain more in the next few sections, uses golang:1.11.1-alpine, and can be changed independently.

If test has a requirements.txt file in the current directory, the dependencies will also be installed as part of the docker image build automatically for both PyPy and Python images. If the test is a Go file, and it depends are external dependencies, these will be automatically installed as part of the Go docker image build. All of the of the dependencies, where applicable are version locked as much as possible, so as to further increase reproducibility.

Although this framework currently only supports Python and GoLang, it could easily be extended to support more languages and frameworks dynamically.

Each image is tagged based on the problem it's running, the language and the solution it's running, determined by the filename (minus the extension).

When the docker image is built, it's constructed in such a way to have as much possible caching as possible. For example, the baseline test is included in every image, and is built first, so that container will only be built once, and can be cached on all other images. Likewise, since all images are built from a specific GoLang image, or a Python (or PyPy) image, they can likewise be cached, and I don't have to wait to fetch multiple images. If I did want to test with multiple versions of a framework, the architecture is such that I could easily expand my testing to include that. Due to time limitations, this is not implemented at this time.

If the docker image fails to build, that test is skipped. This is so that if the last test fails, you don't have to wait for every other test to re-run just to check that result using this framework. In the future, I might add the ability to skip over already run tests, based on their output files, more detail to follow.

Docker will automatically cache docker images. This means that if you re-run a test, and that particular problem file has not changed, docker will use the cached version. For every docker image built, representing a new test, a set of standardized operations are executed. First, a baseline test is run before very test. This test checks for ~1000 prime numbers, written in GoLang. The result of this is not important, I just needed a test that would take long enough to have be able to measure smaller margins of error, and determinant.

If, for whatever reason the baseline fails (exit code not 0), it is automatically retried until it passes. The time taken for this baseline is recorded as *before_baseline*. The main test is then run in the same image, but a new container. The test exit code is also recorded, and the test is retried if it fails (exit code not 0). The baseline is then run again after the test, and the time taken is stored as *after_baseline*. The percent change in baseline scores is calculated, and if there was more than a 5% difference between the two baselines, the test result is discarded, the system waits 10 seconds to give the host a chance to settle, and the retries the iteration starting with the first baseline test. Upon a successful run, meaning there was less than 5% difference between the two baseline tests, stats are collected and stored about that iteration, and the next iteration is started. Every baseline and test container are removed after every iteration to prevent any errors resulting from cached values or partially started containers between runs. While the test is running, I use the docker API to track system stats such as memory usage, cpu usage, and per-core usage. All of that collected data is stored for every iteration. In addition, I also store the time taken for the test itself (time between the container starting and the container exiting), as well as the average time taken for the baseline tests. The test itself is the only process running in the container, so the time taken should be a good representation of how long the test took.

The user is able to specify how many iterations to run every test. For every iteration, this entire process is repeated (the image is only built once), until there are 10 ‘successful’ iterations completed, and data collected.

Once the iterations are completed, I do two passes of data aggregations to be used later for the charts shown below in this report. The first looks in-depth at CPU and memory utilization across a single run (taken from the first iteration of that test). This allows me to plot CPU and memory usage as the test is running incase there is anything interesting about it. The second pass looks at a more general overview, looking at the average results across multiple runs of the same test. This should help bring forward any significant changes between multiple runs of the system. I also store the ‘normalized’ test time against the baseline. This means that I can run the system on two different machines with very different performance or configurations and get similar normalized results.

All of this data is written to two CSV files, again automatically generated, and labelled by the problem set, the interpreter and the solution being tested. This should allow for more languages and frameworks to be tested in the future. There is also much more data that docker collects that I am not currently looking at for time sake. If I were testing distributed system components, I could also look at network usage through docker stats.

In order to help with troubleshooting and repeating tests multiple times, I added a AUTO_SKIP option. This option saves the configs specified from last time. If the configs have changed, it will automatically re-run all the tests, as the user can expect different results. If the configs have not changed, it will automatically skip any tests that already have output in the form of both CSV files mentioned above.

While I am pretty happy with this implementation as a first pass, I have a couple problems that I’d like to address if I had more time. The first, is that I am only running one test (and one

iteration) at a time. This means the testing is very slow, especially to collect many samples of each test. The second is that some of the tests run very quickly, so the docker stats don't always collect data before the container exits. I know you can pause and unpause docker containers, meaning I could pause a container, collect stats, and then unpause the container. This would add complexity to timers and add significant complexity to the testing framework, so for now, this is not implemented.

Problem 1: Barbershop

The barber shop could show up in computer science in problems where you have a pool of workers (threads, or load balanced machines), and some unknown number of clients, each of which are making requests to the system at different times. This pattern is very common in web sites, where users will make different requests to one or more servers, where there are generally less servers than there are clients at any given time.

You can see my the implementations I'll be comparing here: https://github.com/mattpaletta/Little-Book-Of-Semaphores/tree/master/problems/barbershop

The Python implementation is taken from: https://github.com/bragisig/python-sleeping-barber
The Go implementation I wrote myself.

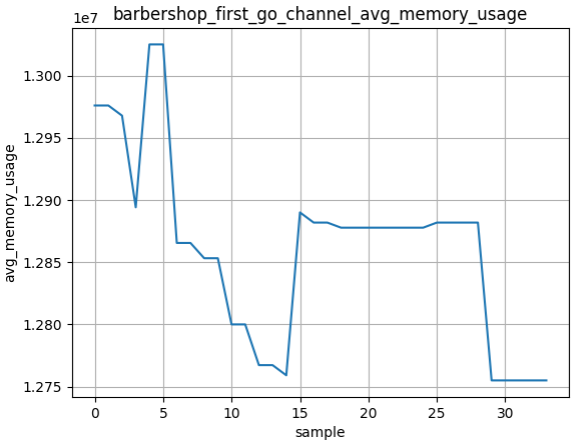
The Python implementation uses a global mutex to lock and unlock when customers are sitting down/getting up. The implementation is event-based. The barber has an internal state where he is either awake or asleep. The customer can wake him, and he will cut their hair, if there are no customers, he will go back to sleep.

The Go implementation instead uses a single channel that is shared between barbers and customers. When a customer arrives, they put some work into the queue. The queue has a certain size, representing the number of chairs in the barbershop. One or more barbers reads from the queue, and pulls events off the queue and does some work on them. If a customer arrives, and the channel is already full, they must wait to put something into the channel until a time when it is available. Likewise, if the barber is looking for more work, he will wait until there is work available before continuing. This behaviour could also easily be translated to Python wherein there is a global thread-safe Queue, and threads are able to read and write from that queue.

	Go (channels)	Python
Correctness	-	-
Comprehensability (max 10)	8	6
Performance (Normalized)	34.402 s	inf
Avg Memory	12.904 MB	-
Max Memory	50.995 MB	-

I would argue that in terms of comprehensibility, I think both solutions are comprehensible, but the Go one is more so. In the Go implementation, both parts, the customer and the barber simply have a queue of input and a queue of output. They are not concerned with the implementation of the other 'side'. In Python, I think it's good that the programmer abstracted

out the coordination as much as possible by having an event state within each class. This unfortunately means that a new developer has to read much more of the source code to understand the interaction, whereas in Go, a developer could have an easier time just abstracting out the rest of the system and just working on those couple functions. The Go implementation also makes it much easier if you wanted to add a third step in the future, such as a cashier, etc.



The Go implementation simplifies the problem by not having a waiting room, and using the size of the channel as the 'chairs'. Nowhere in the code are they explicitly defined. This makes certain small details of the problem easy to miss, or to loose track of. That being said, the Python has a deadlock in it wherein if the customer arrives too slowly, the barber will go to sleep and never wake up. Unfortunately, this meant I couldn't test it as it never terminates. So if the file detects it's being run in a docker container, it exits immediately.

The figure shown on the left shows the average memory usage every second during a single run of the Go implementation. For 100 customers, 1 barber, and 3 chairs, this test took approximately 34 seconds.

The first 5 seconds are likely when Go is creating the channels and the list of customers. The memory usage likely drops as the work is being divided between processes, as the initial queue is decreasing in size. The flatline between 15-28 s is likely when work is being done. The initial incline is as the first batch of work is being taken off the channel, and the flatline is because every time a piece of work is completed, the next one is consumed.

Since the Python implementation has a deadlock in it, and does not terminate, I was not able to collect metrics from it using my automated testing.

Problem 2: Dining Philosophers

The dining philosophers problem appears in peer-to-peer software, where every peer is dependent on one-or-more other peers to accomplish some task as a group. More specifically, this appears if the peer-network is performing some synchronous task. If they are asynchronous, it's a different problem, as the dependencies between peers are much looser.

You can see the implementations I am comparing here: https://github.com/mattpaletta/Little-Book-Of-Semaphores/tree/master/problems/dining_savages

The implementations are taken from: https://github.com/soniakeys/LittleBookOfSemaphores/tree/master/4-Classical_Problems/4.4-Dining_philosophers, and adapted as needed.

Both solutions are written in GoLang. The first, inspired by the book uses semaphores, the second uses a set of mutexes and a global WaitGroup.

The book solution uses Semaphores. The semaphores are custom implemented, and available from the source repository. When a philosopher wants to eat, he signals the philosopher to the

right and to his left. He waits until they both give him the forks, then he eats and releases the forks.

The mutex solution uses locks on each hand, so as to handle coordination between two neighbouring philosophers. When a philosopher wants to eat, he locks both his hands, eats and releases his hands. This is very similar to locking and unlocking forks, except the has no knowledge of who is sitting next to him. He simply waits until his own resources, his hands, are available, uses them, and then releases them for others to use.

	Go (book)	Go (mutex)
Correctness	X	X
Comprehensability (max 10)	7	8
Performance (Normalized)	1.044 s	0.827 s

I believe both solutions to be correct, although there is randomness in the nature of the problem, so it is very difficult to evaluate correctness for this problem specifically.

For comprehensibility, I think they are both very good. I'd say that the book implementation is the more traditional way of thinking of this problem. So for that reason, I'd say it's comprehensible. On the other hand, the mutex option puts all of the relevant code in one place. So the load on reading the code, if you are not familiar with the problem is lessened. For this reason, I give the mutex option a slightly higher score for comprehensibility.

For this problem I used 5 Philosophers with 4 bites on every serving. The implementation from the book took 1.044 seconds and the implementation using mutexes took 0.827 seconds. Since my testing framework only collects data once very second, I was unable to collect metrics for this problem.

Problem 3: Producers-Consumers

This is one of the most common patterns in an event-based system. A very common example is in data collection, where data comes from a bunch of clients into a central queue or bus system, that readers then 'consume' off of.

Both implementations are written in GoLang, and are adapted from: https://github.com/soniakeys/LittleBookOfSemaphores/tree/master/4-Classical_Problems/4.1-Producer-consumer

The first implementation, inspired from the book, uses a wait group to coordinate events back and forth, as well as buffer with a mutex on it to read and write from the queue. The second implantation uses a channel between the producers and consumers. In order to determine when all events have been processed, a wait group is signalled every time an event is processed. The main thread simply waits until enough consumers have finished processing work before exiting.

	Go (book)	Go (channel)
Correctness	X	X

	Go (book)	Go (channel)
Comprehensability (max 10)	4	9
Performance	0.53409 s	0.52811 s

While both solutions are correct, I believe the channel implementation to be much more robust and easy to understand. There is a single shared channel of a fixed size. The producers create events and push them onto the channel, and the consumers read from the channel. By specifying a maximum size of the channel, producers will only produce when there is room in the queue, and consumers will automatically wait until there is an item waiting to be consumed.

The two implementations tested, both written in Go performed within a margin of error of each other, across multiple runs of the system. For this reason, I would choose the channel implementation as it performs very similarly and is much more comprehensible.

Problem 4: Thread-safe queue

Thread safe queues are very applicable to general software problems as a way of handling sudden spikes or drops in load. The queue may be distributed across multiple machines, or may be contained within a single instance, having multiple readers and/or writers to the queue.

For this problem, I wanted to do a three way comparison between Go Channel, Python thread queues and Python process queues. The difference in Python that one spawns multiple threads, sharing a single interpreter, the other with multiple processes, each with their own interpreter.

In all cases, I insert 1000 items into the queue, and each item performs the sum from 1 to 10000. I wanted to have very simple test that would mitigate some of the overhead of putting things in and out of the queue, and instead measure the performance when the system is actually doing some work on each item, as would be the case in most actual implementations.

	Go (channel)	Python (threads)	Python (process)
Correctness	X	X	X
Comprehensability (max 10)	9	9	9
Performance (pypy3, python3) normalized	0.52514 s	(3.945 s) (55.150 s)	(12.115 s) (20.817 s)
Avg Memory (pypy3, python3)	2.29 MB	(55.35 MB) (12.47 MB)	(132.47 MB) (41.44 MB)
Max Memory (pypy3, python3)	51.87 MB	(58.62 MB) (12.88 MB)	(381.25 MB) (66.32 MB)

This test was more of a test of performance, than anything else. All implementations start with the items in the queue, then iterate over the list, performing a map. I was more interested in how to do it in Python, specifically what the performance gap would be, if any between using Python processes vs. threads, which all share a single GIL and interpreter.

Based on the conclusion, it seems that Go is the fastest by far. Second to that, it's between PyPy and python, both running the same code. The PyPy interpreter seems to perform significantly faster with threads, because it removes the GIL, but it requires much more memory to do the JIT compilation at runtime. However, with processes, it requires multiple PyPy interpreters, each of which has a startup time, and takes up more memory than another Python interpreter. Not documented here, but in my own testing, I noticed that PyPy will perform better as you re-run the system within the same environment, as it can reuse optimizations from the last run. I wanted to test the first-run case, so I create a new environment on all platforms before every run.

Problem 5: Reusable Barrier

Reusable barriers are very useful for situations where you need to group work together. It allows a system to easily batch requests together. A reusable barrier could be used with a thread-safe queue to group tasks together coming off of the queue and do some processing on them at once.

I am comparing two implementations, the first is written Go, the second in Python. The implementation in Go is adapted from: https://github.com/soniakeys/LittleBookOfSemaphores/tree/master/3-Basic_Synchronization_Patterns/3.7-Reusable_Barrier

The Go implementation uses a semaphores and mutex to count how many items are in the queue currently, before unlocking to let that group of work through. The python implementation uses a look over a yield, at stores those results in the barrier, before releasing them all at once. This effectively does the same thing, while being more understandable and much less code.

	Go (book)	Python (yields)
Correctness	X	X
Comprehensability (max 10)	6	9
Performance (pypy3, python3) normalized	0.50789 s	(0.1158 s) (0.2971 s)

I have found both solutions to be correct. I think that the Python implementation is much more comprehensible, because you're looking at much less code. It does however require the programmer to be comfortable with yield in Python, though most experience Python programmers will be. I also believe the Python implementation is more reusable, because it's a much more high-level implementation and can easily be changed or adapted to different situations.

Interestingly, both implementations of Python (PyPy and Python) perform better than the Go implementation. All of the runs were less than 1 second long, so I was unable to collect memory or CPU data from these runs.

Problem 6: Generator Merge Sort

For my final problem, I wanted to do something different. I read the blog post from coursespaces: https://medium.com/@_orcaman/when-too-much-concurrency-slows-you-down-golang-9c144ca305a

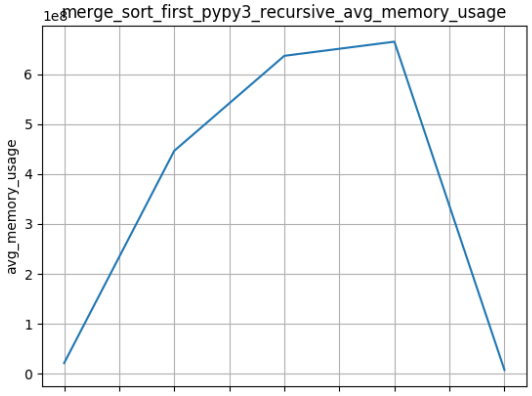
Matthew Paletta Oct 11, 2018

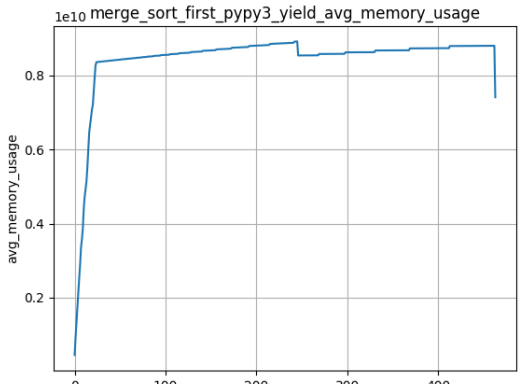
I wanted to try and do it using recursive generators in python. I implemented the generated merge sort myself, and I adapted the recursive adaptation from here (specifically the improved version posted): https://codereview.stackexchange.com/questions/154135/recursive-merge-sort-in-python

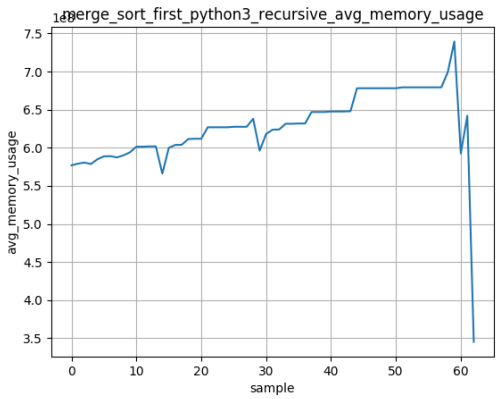
	Python (recursive)	Python (generators)	Baseline
Correctness	X	X	
Comprehensability (max 10)	9	7	
Performance (Pypy3, Python3) normalized	(9.1174 s) (126.45 s)	(904.97 s) (348.391 s)	-
Avg Memory (pypy3, python3)	(355.49 MB) (625.65 MB)	(8449.27 MB) (8861.05 MB)	(80.72 MB) (324.25 MB)
Max Memory (pypy3, python3)	(544.34 MB) (739.57 MB)	(8917.86 MB) (13793.56 MB)	(84.87 MB) (328.58 MB)

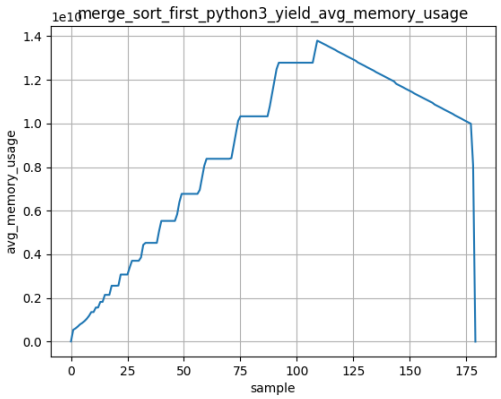
The generated version uses a recursive yield statement in Python, which slowly propagates the next element up the list, until it is no longer the smallest element. This means that if you want to sort the list, but iterate it slowly, this may be a better alternative, as you only compute the next element when you need it. In my tests though, I compute the entire list, as to make it a fair benchmark between the two implementations.

Although I think the generators are very neat, the recursive implementation is more legible, and is more standard. In addition, it uses tail recursion, so if the interpreter or compiler can optimize for that, it could use much less memory. There is probably a way to only have one version of the list store in memory with the generators, but I did not investigate that path.









I found the merge sort to be the most interesting results from my testing. They also took the longest, so I have the most data about them. Both implementations were sorting 10,000,000 integers in Python. Both PyPy and Python use the same code for both the recursive and generated implementations.

For my testing, I first gather a baseline of how big the list of 10,000,000 elements is. PyPy clearly does some optimizations on arrays, as it only took 84.87 MB worst case, whereas Python took 328.58 MB.

As shown in the table above, PyPy and Python are both clearly optimized for recursive functions over generated ones, both in terms of memory and CPU time. This is opposite of the general discussion around Python generators, where they are supposed to be more memory efficient, because they stream in data as it is being processed, presumably so it has a higher chance of still being in CPU cache for the next function.

Looking in the figures for PyPy and Python in the recursive implementation, it seems that PyPy is slowly building one side of the list, then the other, and possibly doing tail recursion. Python on the other-hand, seems to be doing garbage collection as it runs. This would partially explain the longer runs on the base python interpreter.

The story is almost reversed when using generators, where Python seems to build the generators slowly, and PyPy is building them all at once, and keeping them in memory for the entire execution, until they both finally clear them at the end.

I should acknowledge that the PyPy recursive implementation shown only lasted 4 seconds (unnormalized), so there are only 4 samples collected. It is possible that a deeper dive into the execution would yield different results.