Describe in detail your system and the problem it is trying to solve.

The goal is to find the cheapest book and details about that cheapest book for different genres of books from http://books.toscrape.com/catalogue/category/books_1/index.html. The program is designed to scrape the title, description, price, star rating, and availability information of individual books from that site.

There are four Golang packages to accomplish this, in my project:

1. scraper: The scraper package is the powerhouse package. It contains scraper.go which implements the sequential and parallel versions of the program by reading standard input and printing to standard output. The standard input is one of three text files that contains relative url paths to genre links from the website: http://books.toscrape.com/catalogue/category/books_1/index.html, which contains information on different books by genre. There are three standard input files: genres1.txt, genres2.txt, and genres3.txt. They differ by having different genre links to read in. The genres1.txt file is the smallest because its genre links have fewer individual book urls to to scrape than the other input files. The genres3.txt file is the largest - it contains all genre urls from the website. These files are all located in the scraper package in the directory “in-files”.

   The standard output from scraper.go is a text file containing the cheapest book in each book genre. These are saved (or will be saved when running the program) as genres1_Out.txt, genres2_Out.txt, and genres3_Out.txt, respectively, for each problem size in the “out-files” directory. Also in the “out-files” directory is a directory named “answers”. The answers directory houses the answer keys, genres1_Out_Ans.txt, genres2_Out_Ans.txt, and genres3_Out_Ans.txt. These files have the expected output to compare the program’s output against.

   In the scraper package is also a program called check.go, which compares the program output to its respective answer key. If the files match (i.e. the cheapest book returned for each genre is the same) then the program prints “Ok!”. If the two files have a different number of books, the program prints "FAIL!: Answer and output are different lengths” and then prints those lengths and exits. If the two files have the same number of books but one of the books from the program’s output is not in the answer key, then it prints “FAIL! {book} not in answer” and exits.

   Also in the scraper package is timer-and-plot.py and proj3-slurm-job.job for running the program to create the speedup graph. The speedup graph is also contained in this package as “benchmark.png”. See the “How To Run” section, below, for more information on these.

2. book: The book package contains only one program, book.go. The program book.go has the structure of the book object, including the title, description, price, availability information, and star rating. The book.go program also has a method called NewBook to instantiate a book object. The book structure and NewBook method are publicly accessible because both the html-wrangler and linkedlist packages need to be able to create book structures.

3. stealing-queue: The stealing-queue package contains only one program called queue.go. This program contains the data structures and methods to implement a work stealing queue. The work stealing queue is represented as a slice of tasks. There is a bottom int attribute to know what the bottom index of the queue (i.e. the last index) is and also a top attribute to represent an AtomicStampedReference. The top attribute of the queue is a data structure consisting of an index int value and a stamp int value. To create a new stealing queue one could call the NewQueue() method from the package, which instantiates an empty stealing queue. There are three other methods associated with the stealing queue - PushBottom, PopTop, and PopBottom. PushBottom adds tasks to the queue at the next available index position. PopTop removes tasks from the beginning of the queue (i.e. index 0 of the queue’s tasks attribute) whereas PopBottom removes tasks from the end of the slice. I explain more in

the “Advanced Features Description” section how these methods work. Also in the stealing-queue package is a function called NewQueueSlice that initializes a slice of stealing queues. Each entry in the slice will ultimately represent the stealing queue of one of the goroutines. The shared slice structure allows for goroutines to steal from other goroutine’s queues.

4. linkedlist: The linkedlist package contains only one program, linkedlist.go. This is a unique data structure I created to mimic a lock-free dictionary with methods used for the purpose of this project. A node in the linked list contains a key (which will ultimately be a genre name such as “travel”), a value (which is a book structure from the book package), and a next attribute, which points to the next node in the linked list. There are methods in this program to create new nodes and a new linked list. The linked list’s nodes are ordered by the key, alphabetically. So, for instance, historical fiction would be before historical non-fiction. Each node should contain a value of the cheapest book for the genre key.

   There are three methods: Add, Update, and Print. The Add method adds a node to the linked list in a lock-free manner. If the genre (key) is already in the linked list, the program calls the Update method to determine if the new book for that genre has a lower price than the current book for that genre. If it does, it updates the book for that genre in a lock-free manner. See the “Advanced Features” section, below, for more information on these methods. Finally, there is a Print method to visualize the linked list and print to standard output.

5. html-wrangler: The html-wrangler package contains only one program, html.go. The program html.go is responsible for all requests to urls and all web scraping. I use “net/http” to make http requests to urls and then “golang.org/x/net/html" for web scraping. The GetBodyResponse method takes in a url, makes a GET request to that URL, which returns the body response (i.e. the underlying html code for the url) for parsing. The remainder of methods in this program deal with web scraping. In general, what these methods do is recursively parse the DOM of the body response, where the recursive calls are to child nodes and sibling nodes. In these calls, the program is looking for specific html elements and extracting the text or attributes from those elements. See “golang.org/x/net/html" for more information on how Golang parses the DOM of a url.

   The methods GetBookUrls, ParseHtmlForBookLinks, and Next are responsible for finding in the DOM the links to the individual book pages. For instance, if the genre was Travel and the program is scraping the DOM of http://books.toscrape.com/catalogue/category/books/travel_2/index.html, it wants to find the urls of all the books on this page so that we can navigate to those pages and scrape the book detail information. The GetBookUrls method takes in the body response of the GET request from the genre url and calls ParseHtmlForBookLinks on the parsable body response. In ParseHtmlForBookLinks, the function is looking for element nodes in the DOM that are <h3></h3> because the first child node of the <h3></h3> elements are the links to the individual book pages. The program grabs those urls and appends them to a slice for processing, later. The GetBookUrls method also calls Next to determine if there is a “next” button on the genre page such that we need to navigate to that page, do a GET request to get the HTML body for that page, and repeat GetBookUrls and ParseHtmlForBookLinks on that next page.

   The methods GetBookDetails and ParseHtmlForBookDetails are used to parse the individual book pages for each genre, found by the GetBookUrls, ParseHtmlForBookLinks, and Next methods previously mentioned. These work similarly to GetBookUrls and ParseHtmlForBookLinks methods except the ParseHtmlForBookLinks is searching for the attributes associated with the book details (i.e, title, description, price, availability, and star rating). When those details are found, they are put in to a book structure and returned.

Sequential Implementation

In the sequential version of the program, each genre url from standard input is read in one by one. Each genre url is scraped using the html package to find the individual book page urls. As stated before, these individual book page urls are appended to a slice. After all book urls for a genre url are appended, the sequential program loops through the slice, using the HTML package to navigate to the individual book page and scraping the book details, as explained above. The cheapest books returned from the web scraping are added to a dictionary where the dictionary key is the genre and the value is the book data structure representing the cheapest book for that genre. When the book details are returned from the html package, the program determines if the price of the book is less than the current cheapest book in the dictionary for that genre. If it is, then the program replaces the value of the genre key in the dictionary with the book data structure of the cheaper book. When all of the book urls are scraped for a genre url and the program has the cheapest book in the dictionary, the next genre url from standard input is read and the process repeats. When all genre urls and individual book pages associated with those urls are scraped, the program loops over the dictionary keys and values and prints them to standard output. These represent the cheapest books in each genre. The output looks something like this, for instance:

SUSPENSE
Title: Silence in the Dark (Logan Point #4)
Description:
Two years ago, Bailey Adams broke off her engagement to Danny Maxwell and fled Logan Point for the mission field in Chihuahua, Mexico. Now she's about to return home to the States, but there's just one problem. After Bailey meets with the uncle of one of the mission children in the city, she barely escapes a sudden danger. Now she's on the run--she just doesn't know from w Two years ago, Bailey Adams broke off her engagement to Danny Maxwell and fled Logan Point for the mission field in Chihuahua, Mexico. Now she's about to return home to the States, but there's just one problem. After Bailey meets with the uncle of one of the mission children in the city, she barely escapes a sudden danger. Now she's on the run--she just doesn't know from whom. To make matters worse, people who help her along the way find themselves in danger too--including Danny. Who is after her? Will they ever let up? And in the midst of the chaos, can Bailey keep herself from falling in love with her rescuer all over again?With lean, fast-paced prose that keeps readers turning the pages, Patricia Bradley pens a superb story of suspense and second chances. ...more
Price: 58.33 Pound Sterling
In Stock?: Yes
Stars: 3

Parallel Implementation

The parallel version reads in the number of threads flag and sets GOMAXPROCS to that number. It then instantiates a linked-list dictionary (explained in the “Advanced Features Description” of this report) for goroutines to write to as they scrape web pages looking for the cheapest book in each genre. Much like proj2, the program then spawns a go routine to be solely responsible for reading genre urls from standard input and sending those genre urls to a genreUrls channel for processing by worker goroutines. The main function then spawns numOfThreads worker goroutines to read from the genreUrls channel.

Inside the worker function, the goroutine determines if there are more genreUrl tasks in the genreUrls channel. If there are, it grabs one and calls to the html package to parse that genre url in order to find the individual book urls (exactly as in the sequential version) for that genre. See above for the description of the html package for exactly how this happens. As discussed above, the result of this is a slice with the book urls for the given genre. Those book urls for the genre are put in to a bookUrls channel that is local to the goroutine worker. Then, numOfThread more goroutines are spawned to read from that bookUrls channel and scrape the book detail information from the book urls for the given genre. Immediately after spawning all of the goroutines, the worker tries to grab another genre url and repeats the process.

Meanwhile, in the bookDetailWorker function, the spawned goroutines read from the bookUrls channel. Each goroutine spawned has its own stealing queue inside a slice of stealing queues where the slice of stealing queues represents all goroutines' stealing queues. A book detail worker will grab book urls from the bookUrls channel and "PushBottom" the book url to its respective stealing queue. The goroutine will do this until there are no more book urls to read from the bookUrls channel. Note that all book detail worker goroutines are doing this at the same time. Once the book url channel is empty, the goroutines begin to process the book urls in their stealing queues by calling the PopBottom method, which produces a book url for scraping. The goroutine will scrape the data from the book detail page and then call the linked list's Add method to add or update (if applicable) the scraped book details to the dictionary for the given genre. See the "Advanced Features Description" section for more information on the lock-free linked list's (dictionary's) structure and methods. A book detail worker will repeat this process until its stealing queue is empty.

When a goroutine's stealing queue is empty, it notifies other goroutines that there are no tasks to steal from its queue with a flag variable and then atomically updates a shared variable called nothingLeftToStealCount, which represents the number of goroutines that have nothing in their queues (i.e. all book urls the goroutine grabbed have been processed). When a goroutine has nothing left in its queue, it tries to steal from other goroutines' queues, which it accesses through the shared slice of stealing queues. It does this by generating a random index number between [0, numOfThreads] and looking at the stealing queue at the index it just generated from the slice of stealing queues. If the flag for that stealing queue is set to false, then the goroutine knows there is nothing to steal and tries again (i.e. generates another random index and looks to see if that queue is empty or not). When the goroutine finds a stealing queue where there is something to steal, it grabs a url from the top of that stealing queue using the PopTop method. Note that between the time a goroutine checks to see if there is something in the stealing queue and the time it actually grabs the url from the stealing queue, the state of the stealing queue can change such that there is no longer something to steal. This can occur if another goroutine is concurrently stealing or if the queue's goroutine took the last book url from the queue after the stealing goroutine checked to see if there was something to steal. Therefore, the goroutine always checks if the url it got is the sentinel url, which indicates that there is nothing left in the queue. If it gets the sentinel url, it simply tries again with another goroutine's stealing queue (i.e. starts the stealing process all over again). If the goroutine does get a real url, it scrapes the data from it and adds or updates the dictionary (if applicable) and then starts the stealing process over again. Goroutines know there are no more threads to steal from when the nothingLeftToStealCount equals the number of threads. When this happens, all book detail worker goroutines exit.

In the worker function, when there are no more genre urls in the genreUrls channel, the worker waits until all of its spawned book detail worker goroutines to finish processing book urls. When that happens, the worker exits.

Back in the main function, the program waits for all workers to be done and then prints the lock-free dictionary to standard output, which will contain the cheapest book for each genre.

It is interesting to note that the channels in this program are all fan-out. The genresUrl channel is read by multiple workers and the bookUrls channels are read by multiple book detail workers within each worker. There is no fan-in that occurs. At the end of the fan-out (i.e. at the end of a book detail worker) a shared dictionary is simply updated and then after all goroutines return the shared dictionary is printed.

Functional Decomposition: The functional decomposition in this program is splitting up the individual tasks and spawning goroutines to handle each of those tasks. The sequential version gets broken down in to three separate tasks. First, there is a goroutine that simple reads from standard input and puts the genre urls that need to be scraped in to a channel for processing. Second, goroutines are spawned to scrape the genre urls. Finally, goroutines are spawned to scrape the book urls for book details. Therefore, there are three distinct tasks working in parallel: reading

in standard input, scraping the genre urls for individual book page urls, and scraping data from the individual book page urls and adding/updating the dictionary entry for the genre.

Data Decomposition: None.

Advanced Features Description

Lock-Free Linked List (Dictionary)

The Golang dictionary that holds information on the cheapest book for each genre, which is used in the sequential version of the program is not thread-safe. Therefore, it would produce non-deterministic results if concurrent goroutines tried to access it at the same time. Golang has a sync.Map thread safe dictionary, but this is a coarse grained implementation and not efficient from blogs I read on the Internet. Therefore, I decided to create my own data structure, based off of a linked list, to represent a lock-free dictionary.

As explained above, a node in the linked list contains a key (which will be a genre name such as "travel" for this project), a value (which is a book structure from the book package), and a next attribute, which points to the next node in the linked list. There are methods in this program to create new nodes and a new linked list. The linked list's nodes are ordered by the key, alphabetically. So, for instance, historical fiction would be before historical non-fiction. The linked list only has three methods - Add, Update, and Print. I thought about implementing Remove and Contains but I decided against that for 2 reasons. First, my project would not need to make use of those methods - I am never removing a genre after it is added and I am never searching for a genre. Second, after reading The Art of Multiprocessor Programming's section on lock-free linked lists, I determined it would be incredibly difficult to implement Remove because I would have to do CAS on both the marked field and the memory location of the node at the same time. There are ways to do this, but because I don't use the methods I didn't see any point in wasting time to make the methods.

In the Add method, the program traverses the linked list until it gets to the place where the node should be added. If the key value of the node (i.e. the genre) is already in the linked list, the program calls the Update method, which in a lock-free manner updates the value for that node if the new book's price is less than the current node's book price. To do this, the program determines if the new price is less than the old price. If it is, I do an atomic CAS on the current node's value. If the current node's value is where the program thinks it is, then the current node's value is updated to the new book. This CAS can fail if there is a concurrent update happening at the same time. Therefore, if it fails the Update method will try again. When re-trying, it will see if its book's price is less than the current book's price, again, and if it is then it tries CAS again. If there is a concurrent update that prevents a thread from updating the genre's cheapest book, then when retrying the thread needs to go through the entire Update process, again, because the node that was just updated could have a book price that is less than the first node's that was trying to update.

Back in the Add method, if the node's genre key is not in the linked list, the program sets the new node's next value to curr (as we did in all other implementations) and then performs a CAS to try to swing the predecessor node's next value to the new node. The CAS can fail if there is a concurrent Add happening at the same time between the predecessor and current node. If the CAS fails, the program tries again. Note that the concurrent Add may have added a node with the same genre key as the goroutine that just failed. If this happened, then the failed goroutine would call Update. Otherwise, it will try Add, again. This process continues until the thread is successful in either adding or updating a node or exits because its book's price is not cheaper than the current book in the node.

This lock-free linked list (dictionary) makes sure that at least one thread is always doing meaningful work. Because it is lock-free, it is more efficient than the sync.Map built-in.

Work Stealing

In the stealing-queue package, I implemented data structures and methods to allow for goroutines to steal tasks from other goroutines. The work stealing queue is represented as a slice of tasks. There is a bottom int attribute to know what the bottom of the queue (i.e. the last index value) is and also a top to represent an AtomicStampedReference. The top attribute of the queue is a data structure consisting of an index int value and a stamp int value. I think my representation of the AtomicStampedReference is very ingenious. You need the index and stamp value to avoid the ABA problem. However, Golang does not have an AtomicStampedReference built in and there is no way to do a CAS on two different memory locations at the same time. Therefore, we need one memory location for the CAS. This is accomplished with the top data structure. With the top data structure in hand, the program implements the methods in the same way as was presented in week 7.

There are three methods associated with the stealing queue - PushBottom, PopTop, and PopBottom. PushBottom adds tasks to the queue at the next available index position. PushBottom works by appending the bookUrl to the bottom (i.e. last index) of the queue's tasks slice and updates the bottom attribute to point to the next available index position for the next PushBottom. PopTop removes tasks from the beginning of the slice whereas PopBottom removes tasks from the end of the slice. PopTop and PopBottom use CAS atomic operations in order to avoid the ABA problem.

I was initially unsure how to have goroutines steal from other goroutines' queues. How would they get access? Then I realized I could have each goroutine have a stealing queue within a slice of queues that is shared among all goroutines. With this, the index position of the slice represents the go id of a given goroutine. For instance, goroutine 0 would have its queue at index 0, goroutine 1 would have its queue at index 1 and so on and so forth. The goroutines can now access each other's queues. See my parallel implementation explanation above for the details on how the stealing works in the scraper.go file.

The work stealing queue is used by the book detail workers as explained in the first section of this report.

How To Run The Program

NOTE FOR RUNNING LOCALLY: Run the command "ulimit -a | grep open" before you run my program. If you have < 1100 for this number then there is a chance the program could crash from too many webpages being open at once. This is controlled by your OS. You can update the max amount of files allowed for a session with different commands on different OS. I did not want to automate a command in the program, though, to change your computing environment so you would have to do this yourself before running the program. To change this number of a Mac run "ulimit -n 5000" to be able to have 5000 open files at once. You can also just run the program on the cluster with no issue.

Test Cases

Navigate to the "./proj3/src/scraper" directory.

You can test the sequential version by running the command "go run scraper.go < ./in-files/[input file].txt". The input file can be the following: genres1, genres2, or genres3. This will print the output (i.e. the cheapest book in each genre) to the command line. These can then visually be compared to the files in ./out-files/answers" directory.

You can also run “go run scraper.go < ./in-files/[input file].txt > ./out-files/[input file]_Out.txt” where the input file is one of the following: genres1, genres2, or genres3. Then, you can run the command “go run check.go ./out-files/[input file]_Out.txt ./out-files/answers/[input file]_Out_Ans.txt”. This program will print “Ok!” if the program output matches the answer key or a “FAIL!” message if the output does not match the answer key. See information above about the check.go file in the scraper package for more details.

The sequential version takes about 1 min to run for genres1 and about 2 minutes to run for genres3.

You can change the command to be “go run scraper.go -p=[numOfThreads]” to execute the parallel version. The input, output, and checking files will be the same.

I would recommend running the following commands/test cases:

```
go run scraper.go < ./in-files/genres1.txt > ./out-files/genres1_Out.txt
go run scraper.go -p=4 < ./in-files/genres2.txt > ./out-files/genres2_Out.txt
go run scraper.go -p=6 < ./in-files/genres3.txt > ./out-files/genres3_Out.txt
```

These will produce the results to the ./out-files files.

You can then compare your results to the answer key using the commands:

```
go run check.go ./out-files/genres1_Out.txt ./out-files/answers/genres1_Out_Ans.txt
go run check.go ./out-files/genres2_Out.txt ./out-files/answers/genres2_Out_Ans.txt
go run check.go ./out-files/genres3_Out.txt ./out-files/answers/genres3_Out_Ans.txt
```

Or you can just compare, visually. Note that the order of the output from the commands you run and the order in the answer key will not be in the same order. What genres get printed first depends on whether you are running the sequential or parallel version and then for the parallel version, what happens during runtime. However, you can search within the answer key for the cheapest book in the genre to make sure it matches.

Analysis

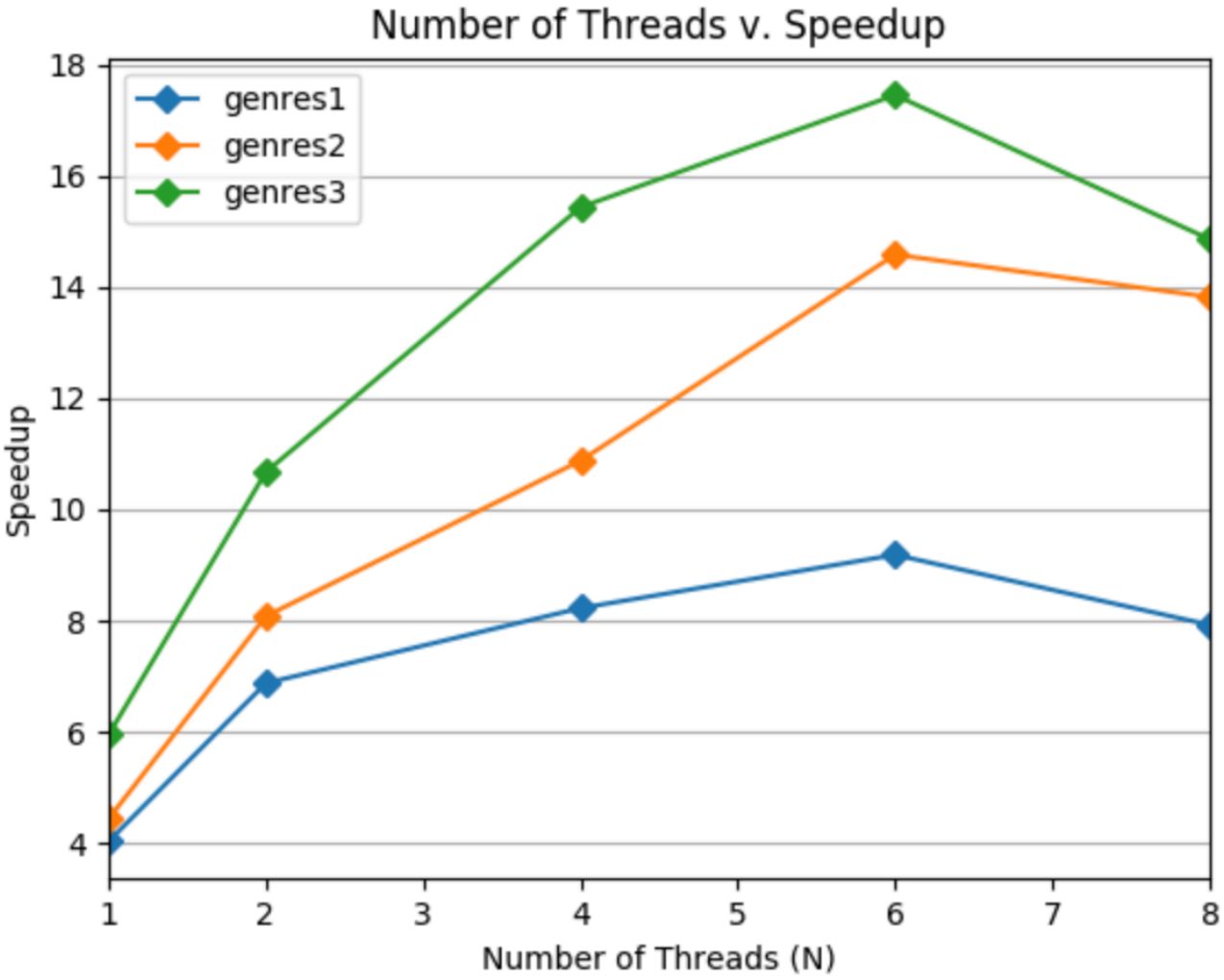
Experiment Description

In this experiment, there are 3 problems of sizes genres1, genres2, and genres3. The problems have different sizes based on the number of individual book pages that need to be scraped. For instance, genres1 has genres with less books than genres2. Therefore, there are less individual book urls to scrape for genres1.

There are 5 thread amounts (goroutines spawned): 1, 2, 4, 6, and 8. For each problem size, the scraper.go program was run such that the number of threads is equal to 1, 2, 4, 6, or 8. There are 15 such timing operations (3 problem sizes * 5 thread amounts = 15 timing operations). The sequential version of the program is run for each problem size (3 problem sizes = 3 timing operations), as well. Speedup is calculated as the sequential version time / parallel version time. So, for instance, if the genres2 took 1, 0.9, 0.6, 0.4, and 0.3 seconds to run for 1, 2, 4, 6, and 8 thread amounts, respectively, and the sequential version took 1.5 second to run, then the plotted speedup would be (1, 1.5), (2, 1.67), (4, 2.5), (6, 3.75), (8, 5) for the genres2 problem size line on the output speedup graph.

For each problem size and number of threads, scraper.go is timed 20 times and the average of those 20 times is plotted in order to avoid misleading timings caused from caching of processes on the first execution, other applications running, or other random interrupts. Therefore, overall, 360 timing results are calculated ((15 parallel timing events + 3 sequential timing events) * 20 iterations for each event = 360 calls of scraper.go).

The above graph is the result of running 20 iterations of each timing event on the general cluster.



Results Explanation

Between 1 and 8 threads I get speedups that range between 4 and 18. The reason that the speedups are so good - like crazy good - is because the sequential version is comparatively very slow compared to the parallel version. At the same time, the sequential version is as efficient as it can possibly be (confirmed by Professor Lamont). The sequential version of the program is heavily bogged down by trying to perform all scraping operations (scraping on over 1000 urls) on one core. The package I use for scraping may be very inefficient in that it goes over the entire DOM recursively to find the html elements I'm interested in. So the computation of scraping takes a "long" time. Compounded on that is the fact that only one genre url can be processed at a time and in that genre url only one book url can be processed at a time. Between the inefficiency of the scraping and the bottleneck of only one url being able to be processed at a time, there is a massive bottleneck in the sequential version. In the parallel version, as I discussed with Professor Lamont, the extensive parallelization by spawning numOfThreads goroutines to process genre urls and then numOfThreads goroutines for every genre url to process that genre's book urls, supersedes and essentially eliminates the inefficiencies of the web scraping package and the sequential bottleneck. For instance, for the genres3 problem size, it takes on average a little over 2 minutes for the sequential version to run, whereas the one thread version takes 22 seconds to run on average. This massive discrepancy indicates that the above theory is likely true, as one would expect the sequential and 1 thread versions to have similar run times, but they do not in this case.

Therefore, it is not surprising that the relative run time of the parallel version is much faster than the sequential version, causing high speedups. Amhdal's Law would suggest that 99.9999999% of my program is parallelized, based on the speedups I get. I believe my code to be highly parallelized, as explained in the first section of this report. There are only minor points of synchronization. However, it is not 99.999999% parallelized. Rather, it is simply the comparative difference between the sequential and parallel versions that is causing the massive speedups. Based on the graphs, though, one can see that more parallelization with more threads leads to higher speedups, indicating efficiencies gained from being parallelized beyond the fact that the scraping package isn't as bogged down.

This was an embarrassingly parallel problem that required almost no dependencies. However, there were some. The program uses channels to coordinate between the functionally decomposed portions of the code. Channels are communicating sequential processes (CSP), which is a method for describing patterns of interaction in concurrent systems. The program specifically uses unbuffered channels which perform synchronous communication between goroutines. These unbuffered channels guarantee that an exchange between two goroutines is performed at the instant the send and receive takes place. If the goroutines are not ready at the same instant, the channel makes the goroutine wait (i.e. synchronization). Another slight dependency was the use of the CAS operation in the linked list dictionary. If a CAS operation failed, the thread would have to try again, causing contention. All of these lead to some slowdown.

With the above implementations, before I added the work stealing queue, my program was actually getting speedups between 5 and 40. After adding the work stealing queue, which required more atomic CAS operations that created slow downs and atomically updating a shared variable to determine what goroutines were finished, the speedups dropped to what they are now. It is interesting to note that for all three problem sizes, the speedups have an upward trajectory as the number of threads increases up to 6 threads but then all decrease at 8 threads. This exemplifies the toll of the atomic CAS operations in the work stealing queue. As you add more threads you have more calls to the atomic CAS operations, creating more contention that could limit some of the efficiencies gained from spawning more goroutines.

Finally, note how the smaller problem sizes have less speedup that then larger problem sizes. This makes sense as, in theory, larger problems would benefit more from parallelization of the program because larger problems have more pronounced hotspots and bottlenecks in their sequential versions. That is, the weight of computation being put on one core is more exaggerated. For instance, genres3 has to scrape a little over 1000 individual urls. Only one url can be scraped at a time, however, in the sequential version. Therefore, the bottleneck is more exaggerated than, say, genres1 which only had about 100 individual urls to scrape. The relief of distributed work for the larger problem sizes when parallelized is observed in the higher speedups for larger problem sizes.

Other Questions

Describe the challenges you faced while implementing the system. What aspects of the system might make it difficult to parallelize? In other words, what do you hope to learn by doing this assignment?

The hardest parts of making this system were implementing web scraping using Golang, creating a unique data structure for the needs of my project, and making the work stealing queue. All three of these things were never demonstrated on other homework or projects. With web scraping, I had to become familiar with Golang's "net/http" and "golang.org/x/net/html" packages for working with http requests and web scraping. This involved a lot of documentation review and really understanding the different structures included in these packages. It was beneficial to learn the Golang web scraping framework, though, as it complements the webs craping algorithms I know in Python.

The lock-free linked list (dictionary) was created from scratch/wasn't discussed in class and allowed me to implement a unique lock-free data structure. In my future career, I may have to make structures beyond stacks, queues, and linked lists that are lock-free. In these situations, I may not have a book to help with my implementation because a lock-free version of the data structure may never have been created before. Therefore, it was useful to think about what needed to happen in order for my dictionary linked list to be lock free and also what concurrent events could make CAS operations fail and then addressing those issues in the code.

The work stealing queue may have been the hardest part to implement. I was confused about how to have goroutines steal from other goroutines when work stealing is already built in to Golang. It was difficult to coordinate this as it required some synchronization. See above sections for more information on how I overcame these issues.

One hard thing about the parallelization is that so many webpages could be being scraped at once, instead of just one at a time like in the sequential version, that the website could put up a block to stop you from querying so much. Additionally, some OSes only allow a certain number of "files" open at once. Opening webpages count as files, apparently. For my computer the number of files I'm allowed to have open at once is 256 ao the program would crash sometimes from my OS complaining that there were too many file open at once. This was resolved when running it on the cluster. Web scraping, in general, when scraping from a "real" website may not be able to parallelized. Many websites have blocks to stop web scraping because they do not want entities stealing their data on a large scale. Thousands of queries in a number of seconds from one IP address raises red flags and can lead the website to block the IP address. Many websites also have Terms and Service Agreements that prohibit web scraping and if the scraped data is being used for business purposes, can lead to legal retaliation. This was not an issue with my project because the website I used was unsecured and was made for the purpose of web scraping.

Specifications of the testing machine you ran your experiments on (i.e. Core Architecture (Intel/AMD), Number of cores, operating system, memory amount, etc.)

I ran the experiments on both my machine and the University of Chicago cluster - general partition. For my machine, the specs are as follows:

macOS Catalina, Version 10.15.5
MacBook Air (Retina, 13-inch, 2019)
Processor 1.6 GHz Dual-Core Intel Core i5
Memory 16 GB 2133 MHz LPDDR2
Graphics Intel UHD Graphics 617 1536 MB

My computer also has multi-threading.

On the cluster, I run my script on the general partition, which has the following specs:

16 Cores (2x 8core 3.1GHz Processors), 16 threads
64gb RAM
2x 500GB SATA 7200RPM in RAID1

What are the hotspots and bottlenecks in your sequential program? Were you able to parallelize the hotspots and/or remove the bottlenecks in the parallel version?

In my sequential version, the hotspots - where the most "computations" are taking place - is the scraping of data from the individual websites. These computations involve traversing the entire DOM of the webpage multiple times, looking for the html elements of interest. The bottleneck occurs because only one genre url is processed at a time. That is, all of the individual book urls for a given genre are scraped for their details, compared to the book currently in the dictionary for that genre, and the dictionary entry is updated if need be before any other genre can go. Additionally, only one book url can be processed at a time in the sequential version. Therefore, there are many urls just waiting to be processed.

There is no data decomposition in my program, so scraping the DOM of websites is not split up in to multiple parts. A single goroutine is responsible for scraping the entire DOM of the website it gets. Therefore, the complexity of the computation is not reduced. However, the functional decomposition allows for multiple urls to be scraped at the same time, which removes the bottleneck of genre urls and book urls having to wait and takes the toll of all of the computations off of one core. Therefore, my program was able to remove hotspots and bottlenecks.

What limited your speedup? Is it a lack of parallelism? (dependencies) Communication or synchro- nization overhead? As you try and answer these questions, we strongly prefer that you provide data and measurements to support your conclusions.

As explained in the "Analysis" section, above, the speedup obtained in the parallel version is very high because 1) the sequential version is very slow and 2) there is very limited synchronization. The only synchronization occurs within the channels and the atomic operations in the lock-free linked list dictionary and the stealing queue. There are no dependencies with the data - which are the book urls - because all book urls can be scraped independently of each other.

I ran interval timings on sections of my code and found that in the parallel version, the section that was accounting for a large portion of the total execution time was when the book detail workers attempted to steal book urls from other book detail workers' stealing queues. For all problem sizes, the time spent by work stealers attempting and retrying the CAS operations needed for stealing accounted for about 45% of the entire program's run time. As mentioned, previously, I ran the program without the work stealing queue and the program's speedup approximately doubled for all problem sizes and thread amounts, which makes sense after identifying that the stealing queue operations were accounting for almost half of the total execution time. For instance, the genres3 problem size for 8 threads had a speedup of 29 without the stealing queue whereas with the stealing version it has a speedup of a little over 15. The next largest chunk of time was just the time it takes to scrape data from the website - which has nothing to do with the parallel implementation. Perhaps it would have helped to have some data decomposition, as well, where goroutines split up the DOM of a webpages and searched independently for the html element of interest. The channels also seemed to work very quickly because there were a lot of goroutines reading from them. Therefore, there must not have been much waiting by the channels for a worker to be ready because the shear number of goroutines spawned essentially would ensure the receiver was always ready.