CS 378H Lab 2 Report

Slava Andrianov

Introduction

In this report, I provide an overview of my findings from implementing and benchmarking a program which hashes and compares binary trees using Go's goroutines and synchronization primitives. I also look into the effects of distributing access to work and data structure modifications and examine their effects on the program's performance, while also demonstrating my ability to construct a custom concurrent buffer similar to Go's built in channels.

Approach

Parsing Input And Creating BSTs

After the input arguments are parsed, the program takes in a path to the input file and reads it all into memory at once, interpreting it as a string. After that, the main thread parses this string, creating an array of integer pairs which denote the start and end indices of each line in the file while also counting the number of BSTs. Afterwards, the main thread will create an array of the size of the number of lines in the file to store all of the created BSTs. Then, an optional command line argument dictates whether the BSTs should be created sequentially or in parallel. If the argument indicates that the trees should be created sequentially, the main thread will then iterate over the array of pairs, taking a slice from the input string with the specified start and end and then parsing that slice to create a BST struct. Otherwise, the main thread will instead spawn a goroutine for each BST in the file and that goroutine will be responsible for taking in the relevant start and end, creating the BST struct, and storing it in the array, with a wait group being used as a barrier for synchronization. The BST struct is a relatively simple struct which stores a pointer to a root tree node struct and a size variable to store the number of nodes in the tree. Each tree node struct is simply composed of an integer to

store the value and two pointers to the node's left and right children. The BST struct also supports an insert method to add an integer as a node in the tree per the BST properties, and a method to calculate the BST hash while simultaneously storing the in order traversal which will be discussed further in the section on creating the hash groups. After this step, there is an array of all of the BSTs from the input file now stored in their BST representation in memory, ready to be used in future steps.

Calculating Hash Time And Hash Group Time

When it comes to computing hash and hash group times, my program first checks if the hash workers flag is set to zero, and if it is then the number of hash workers will be updated to the number of trees, resulting in there being a goroutine spawned for each tree for the hash calculations and potentially the hash grouping. Next, the program checks to see if the data workers flag is present. If it is not, then the program will calculate just the hashes either sequentially or with the specified number of hash worker goroutines and returns early, outputting the hash time and program execution time.

If the data workers argument is specified, then the program will enter one of several configurations depending on the relationship between the number of data workers and the number of hash workers as the program proceeds into the hash grouping stage. For the hash grouping stage, the program initializes a map where the key will be an integer hash, and the value is a pointer to an integer slice which stores the BST ids of BSTs with the same hash as the key. There is also an array of integer slice pointers called the traversal store, which is sized to the number of BSTs. This array stores the in order traversal for each BST. This exists because the calculation of the hash is created from an in order traversal of the BST, which is also used to compare the trees during the tree comparison step. As such, the BST struct has a function which takes in a pointer to an integer slice and simultaneously computes the hash for the BST and populates the input slice with the in order traversal of the BST. This accelerates the program overall as storing a value in the slice right after it is used to contribute to the hash is very cheap and it greatly accelerates the tree comparison stage as the comparison workers can use the previously generated in order traversal instead of needing to traverse the tree and create it again. Another important bit of set up is that for the configurations described below, there is an atomic integer which is atomically incremented each time that a hash worker calculates a hash. When this counter reaches the number of trees, the program takes the current time to be the end of the hash calculation stage, but not the hash grouping stage.

If the number of data workers is one, then the program first creates a wait group initialized to the number of hash workers plus one, and creates a channel which is buffered with a size of the number of trees. Then, one data worker goroutine is spawned, which has an internal counter which starts at zero and counts up to the number of trees using a loop. In each iteration, the data worker takes an integer pair struct from the channel which represents a hash and a BST id. The data worker will then insert the BST into the slice in the map for the specified hash. Once the counter reaches the number of trees, the data worker goroutine will synchronize and exit. The program also spawns the specified number of hash worker goroutines, which are given an initial index as a parameter, and they then loop while that index is less than the number of trees. Inside of this loop, the hash worker allocates an integer slice of the size of the BST at the specified index, and stores a pointer to that slice in the traversal store. That slice is also then passed in to the BST function which calculates the BST hash while storing the in order traversal of the tree in the passed in slice. After this operation is complete, the hash worker then passes the calculated hash and the BST id into the channel so that the data worker goroutine can update the map of hashes to BSTs with those hashes, and it increments its index by the number of hash workers. This loop being controlled by the index being less than the number of hash workers ensures that any number of goroutines will always be able to calculate the hashes correctly for any number of trees and that extra goroutines will exit and not alter the correctness of the program.

If the number of data workers is equal to the number of hash workers, the program creates a wait group initialized to the number of hash workers and a mutex to protect updates to the map. The program then creates the specified number of hash workers which perform the same sequence of actions as the goroutines in the case where the data workers value is one, but now instead of sending the BST hash and index into a channel, they instead acquire the mutex, insert into the map themselves, and then unlock the mutex.

If the number of data workers is less than the number of hash workers, the program will create a wait group initialized to the specified number of hash workers, a channel with its buffer set to the specified number of data workers to serve as a semaphore, and a read-write mutex. Additionally, an extra lock map is created, where the key is an integer hash, and the value is a pointer to a mutex. This structure exists to ensure that multiple hashes in the map can have their lists updated safely without concurrent updates to the list for the same hash. Then, the program spawns the specified number of hash workers which have the same structure as the ones for the case where the number of data workers is set to one, however the logic for inserting into the map is quite different. Once the hash is ready, the hash worker sends an empty

struct into the channel to decrement the semaphore, and then gets a read lock on the read-write mutex, attempts to access the map entry for the relevant hash in the map, saving the error code, and then unlocks the read lock. Then, if the error code indicates that the hash does not already have an entry in the map, the worker tries to get a write lock. Once the hash worker has write access, it checks the map for the hash again, and if it is still not present then the hash worker will create the entry and mutex for this hash to be stored in the earlier mentioned lock map and release the write lock. The reason for this additional check inside of the write lock is to not overwrite the entry and lock in the case that another goroutine had created the entry for the hash in the time between that this goroutine had checked for the entry and obtained the write lock. After this step, or if the entry had been found to be present in the map, the hash worker obtains a read lock, and saves the pointers to the slice of BST ids and the mutex for this hash into local variables on its stack, and then releases the read lock. Since the two maps in question both store pointers to their integer slices or mutexes, this means that it is actually safe to use these locks and slices at the same time that another hash is being inserted into the map, as the map may move where those pointers are stored, but those pointers will still give access to the same place in memory. As such, the read lock ensures that any number of readers can safely concurrently update the slices of BST ids for hashes already in the map, and that if a new hash needs to be inserted then the write lock will block the goroutines which wish to read until it the map is done being updated. Once the pointers have been copied to the stack, the hash worker then locks the mutex that it just got a pointer to, updates the slice that it got a pointer to, releases that same mutex, and then takes an empty struct from the channel as a way of incrementing the semaphore value. Through this combination of the channel acting as a semaphore, the map of mutexes for each hash, and the read write lock, the program allows for finer grain synchronization where the specified number of data workers can safely concurrently update the slices of BST ids with a given hash.

Regardless of the method used to calculate the hash groups, after they are all calculated the main thread then checks whether the comp-workers flag is present. If it is not, then the program prints out the hash groups, the timings for the hashing, grouping, and program time, and exits. Otherwise, the program moves on to the tree comparison section.

Calculating Tree Comparison Time

Prior to the tree comparison section, the main thread initializes an array of booleans the size of which is the number of BSTs in the input file squared. This array is a compressed two dimensional matrix, where if you have some tree with index i and

another tree with index j, the index (i * num_BSTs_in_file) + j is the boolean which will indicate if the tree at index i is identical to the tree at index j. After initializing the array, the main thread notes the starting time of the tree comparison section and determines how to perform the tree comparisons based on the comp workers command line argument that was passed in to the program. In all cases, the main thread iterates over all of the hash groups, and inside of each hash group it tries comparing all combinations of tree indices in the hash group using two for loops. To make it easier to explain the algorithms, I will refer to two trees being compared as tree i and tree j, where i and j are the BST IDs that have been discussed previously. In these loops, the main thread is seeking to populate spot [i][j] in the adjacency matrix, with a boolean indicating if the two trees are the same. If i and j are the same, or if spot [j][i] has been marked as true, the main thread will set spot [i][j] to true and continue, because you are either comparing the tree against itself, or there has been some previous comparison which found that the trees with these IDs are identical. If neither of these conditions are met, then the two trees need to be compared. Since the hash grouping stage stores the in order traversals for each BST in an array, this comparison is simply just accessing the relevant traversal array and seeing if they are the same. However, how this method is executed is determined by the passed in value for comp workers.

If the value for comp workers is one, that corresponds with the sequential implementation and the main thread does all of these comparisons on its own. If the value for comp workers is zero, this indicates that a goroutine should be spawned for each comparison, so a wait group is created and right before each goroutine is launched, the wait group's counter is incremented by one. Each spawned goroutine then performs the comparison between the two BST ids given to it, writes the result of the comparison into the array, and exits after decrementing the wait group counter. If the value for comp workers is greater than one, then the program again creates a wait group to serve as a barrier, and initializes its count to the specified number of comp workers. The main thread also creates a custom concurrent buffer whose capacity is algo initialized to the specified number of comp workers. The implementation of this concurrent buffer is discussed in more detail in the next section, but it functions somewhat similarly to a buffered Go channel which stores integer pairs, where each integer pair notes the ids of the two BSTs which need to be compared. Then, the main thread spawns the comp workers goroutines, which sit in an infinite loop, attempting to take a value from the concurrent buffer. The concurrent buffer is blocking, so these goroutines are blocked when there are no comparisons to be performed. Once one of these goroutines takes a value from the buffer, it first checks to see if the two given BSTs are negative one. It is impossible for a BST to have an index of negative one, so receiving an integer pair of two negative ones is used as a means of telling the goroutine that it should synchronize on the wait group and exit. If this condition is not

met, the goroutine will instead compare the BSTs with the two IDs that it took from the pair in the buffer, write the result to the adjacency matrix, and then proceed on to the next iteration of the loop. After creating these worker goroutines, the main thread iterates over the hash groups and the indices inside of each hash group, except now instead of doing the comparisons itself or spawning a new goroutine for each comparison, it instead inserts the the IDs of the two trees which need to be compared into the concurrent buffer and continues onward. Once it is done iterating over all of the hash groups, the main thread sends through comp workers pairs of two negative ones to indicate to all of the comparison worker goroutines that they need to shut down. Since the concurrent buffer's get method gives out values in the same order that they were placed into the buffer, it is guaranteed that all of the tree comparisons will finish before all of the comparison worker goroutines shut down.

Custom Concurrent Buffer Implementation

My custom concurrent buffer is a struct composed of an integers for the current size, or number of elements in the buffer, the capacity of the buffer, the index that producers will place into, and an index which consumers will take from. It also contains a slice of integer pairs which serves as the data storage for the buffer, a mutex, and two condition variables, one for the producers and one for the consumers. There is an initialization method which takes in an integer for the capacity, and this creates the concurrent buffer struct, initializing it with the capacity, setting the initial size and indices to zero, and creating the slice and relevant synchronization variables. When placing into the concurrent buffer, the mutex is locked and the goroutine attempting to place into the buffer will check if the buffer is already at max capacity, which is safe to do because of the protection of the mutex. This check is placed on a while loop, so if the buffer is at capacity then the goroutine will wait on the producer condition variable, thereby blocking itself and releasing the mutex and allowing another goroutine to proceed with its operations. If the buffer is not at capacity however, then the placing goroutine will bypass the loop and instead be able to place a value into the buffer and update the buffer size and the producer index, with there being some logic to ensure that the index is never going to write to an out of bounds slice index. Once the write is complete, the goroutine signals on the consumer condition variable and releases the mutex. The case for retrieving a value from the concurrent buffer is symmetric, except now the condition for the while loop is if the number of elements in the buffer is zero, and after reading from the buffer the consumer index is updated and the producer condition variable is signaled.

Performance Results

Testing Environment And Procedure

All data presented in this section was gathered on a system running Ubuntu 20.04.6 with the GNU/Linux 5.15.0-127-generic x86-84 kernel. The processor used was a Intel i5-14600 with 20 cores, a 336 Kib L1 data cache, a 224 KiB instruction cache, and a 15 MiB L2 cache. The system also had 31 gigabytes of memory. All code was compiled and tested with Go version 1.13.8 due to this being the default version on the machine that was used for development and data collection.

All timing measurements were made using the time library from the Go standard library. End to end timing was measured by taking the time right after the entry into the main function and the time right before the final print statement which prints the end to end time and subtracting the two. Timing for the hashing, hash grouping, and tree comparisons were started right before the if statement which selects which methodology is used to perform the comparisons, and ended right after the main thread exited the relevant if else block which determined the methodology used. Some more detail on the exact placement of these timings is provided in the approach section, but all placements follow this general rule.

Testing was performed using three benchmarks: a simple benchmark composed of twelve small trees, a coarse benchmark composed of one hundred relatively large trees, and a fine benchmark composed of one hundred thousand relatively small trees. For each of these benchmarks, data was collected using a bash script which first warmed up the system by running the fully sequential version of each configuration ten times. Each of the described configurations was run ten times, and the average of these ten runs was used for the speedup comparisons described here. All of these configurations parallelized their BST construction by spawning a goroutine for each constructed tree. For more detail on this, please refer to the approach section. While I would have liked to have experimented more with different values for the number of goroutines being used to create the trees, this was not a focus of this lab so I elected to always spawn a goroutine for each tree as it generally provided a nice speedup over the sequential tree construction, helping speed up testing and data collection.

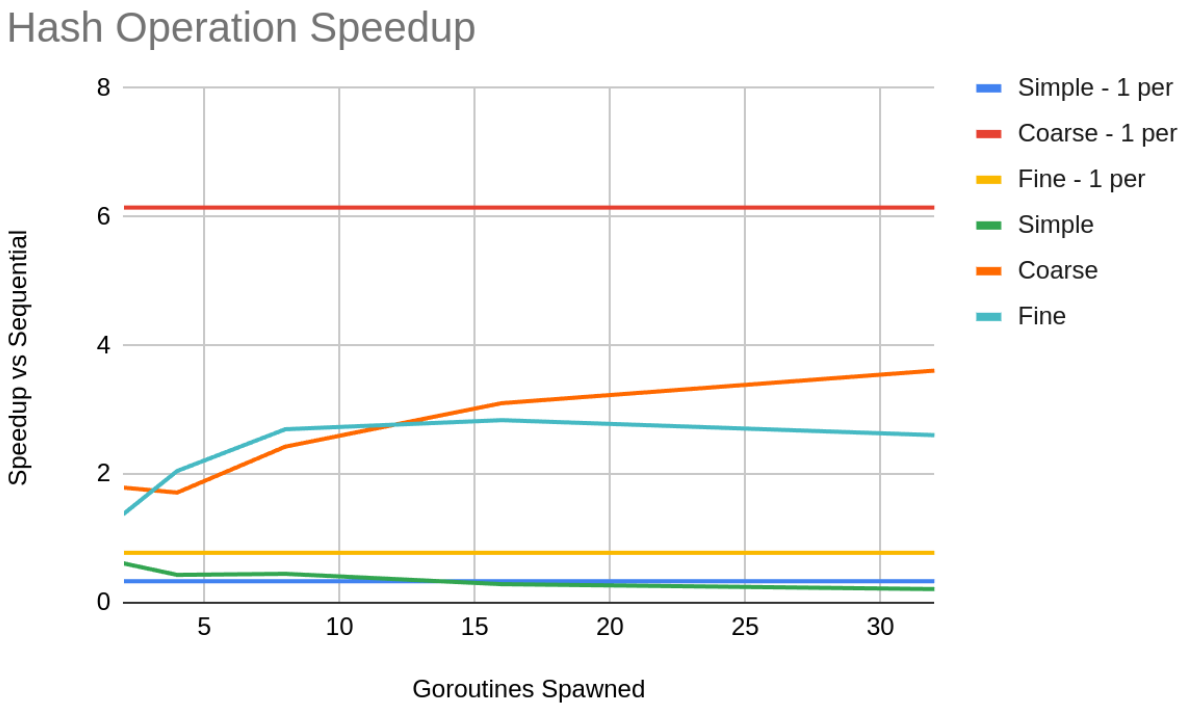
This testing was also conducted using the details from the original spec for the lab, meaning that for the hash calculation, only the hash workers argument was specified, allowing the program to early return right after the hash calculations. Similarly for the

hash group testing, only the hash workers and data workers arguments were specified, allowing the program to return early right after. These early returns ensure that the end to end runtime comparisons are more fairly comparing the effects of the different configurations and not the results of earlier or later steps in the execution. In the case of the tree comparisons, the hash workers and data workers values were both set to one to better isolate the effect of the changes to the tree comparisons.

For all cases other than the extra credit, the number of hash workers, data workers, and comparison workers used for testing were 2,4,8,16, and 32. For the extra credit, The number of hash workers was fixed at 16 as this number of hash workers generally demonstrated a reasonable amount of performance, and the number of data workers was varied from 2 up to 8 to test the effectiveness of the fine grained locking approach.

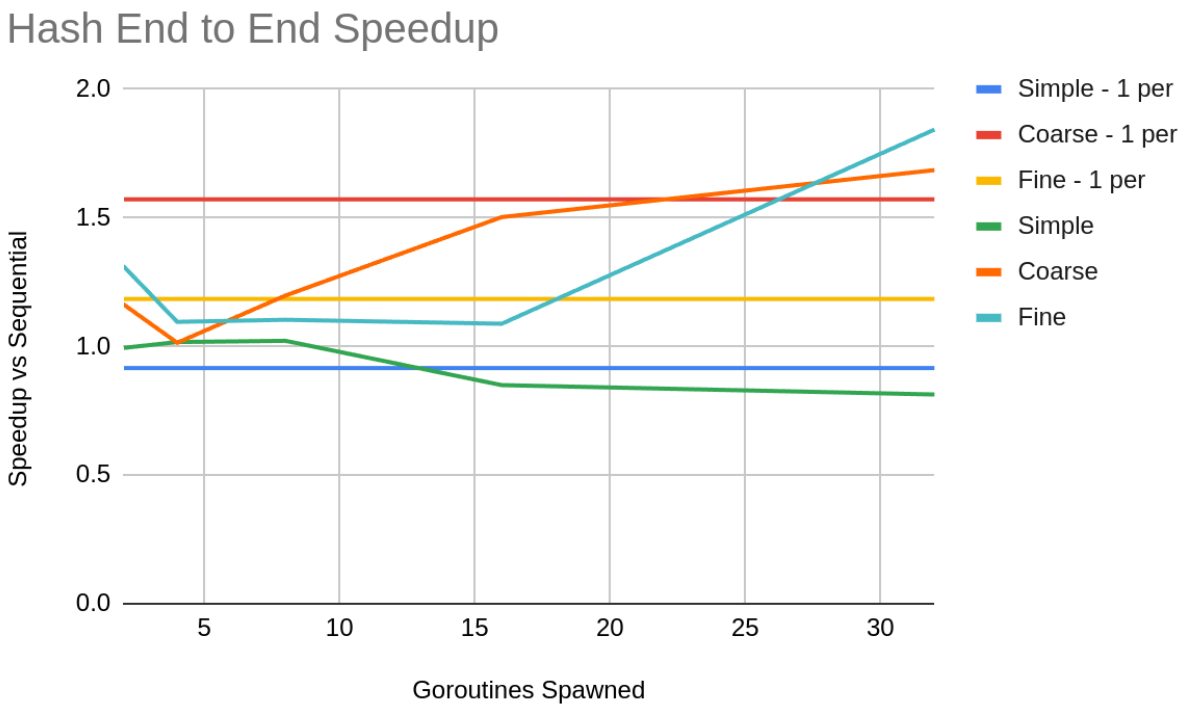
All of the graphs presented in this section are presenting the speedup of each configuration against the fully sequential implementation. An important note is that in the case of a comparison of two sets of results which both fail to achieve a speed up over the sequential algorithm, some result being described has having a greater speedup over another can also be thought of as having the performance of that configuration be closer to that of the serial algorithm, indicating better performance in comparison to the other discussed benchmark or configuration. Another important note on the graphs is that while the one goroutine per tree configurations are shown to stretch across all goroutine counts from two to thirty two, the number used is just the average of the runs, graphed as a single straight line for easier visibility.

Calculating BST Hashes



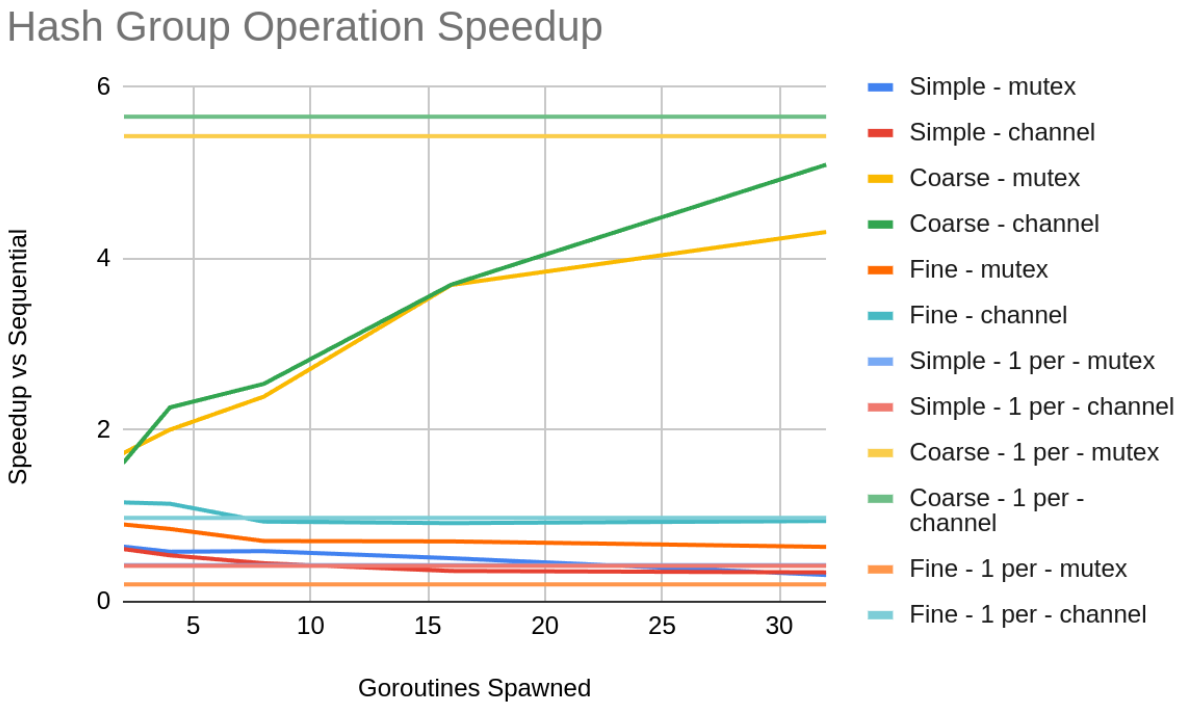
In the above chart, it can be seen that aside from the small benchmark and the fine benchmark with one goroutine being spawned for each tree, all values for the numbers of hash workers produced a speed up for the hashing operations over the sequential implementation. This is inline with expectations, as the simple benchmark is very small in both the size of the trees and the number of trees, so the overhead of creating and managing the goroutines is expected to outweigh the actual work of traversing these small trees. In the case of the fine benchmark, the trees are also relatively small, but given that there are one hundred thousand of them means that there is a very large overhead for creating, swapping between, and destroying all of these goroutines which greatly outweighs the benefit of being able to compare the relatively small trees in parallel. The speedup for the coarse benchmarks and the fine benchmark with a number of goroutines between two and thirty two is also expected. Comparing trees in general is expected to be a somewhat work intensive application as the BSTs are composed of multiple nodes which are likely not adjacent to each other in memory, leading to many cache misses which slow down the operation. As such, there is an advantage to being able to do this work in parallel, especially in the case of the coarse benchmark due to the large trees. The fine benchmark also gains a benefit here because while the trees are fairly small, there is a very, very large number of them, making it advantageous to calculate the hashes for them in parallel so long as there are not so many goroutines that their creation and management overhead becomes too large. The largest speedup for this data was seen on the coarse benchmark while creating one goroutine for each tree, which also aligns with expectations as one

hundred goroutines is still a relatively small number, and the trees in the benchmark are relatively large, meaning that it is able to make the most off being able to perform work in parallel while keeping the overhead of creating and managing the goroutines fairly minimal. Generally, it seems that the Go runtime is able to manage goroutines very well, until the number of them becomes very, very large as is the case with the fine benchmark.

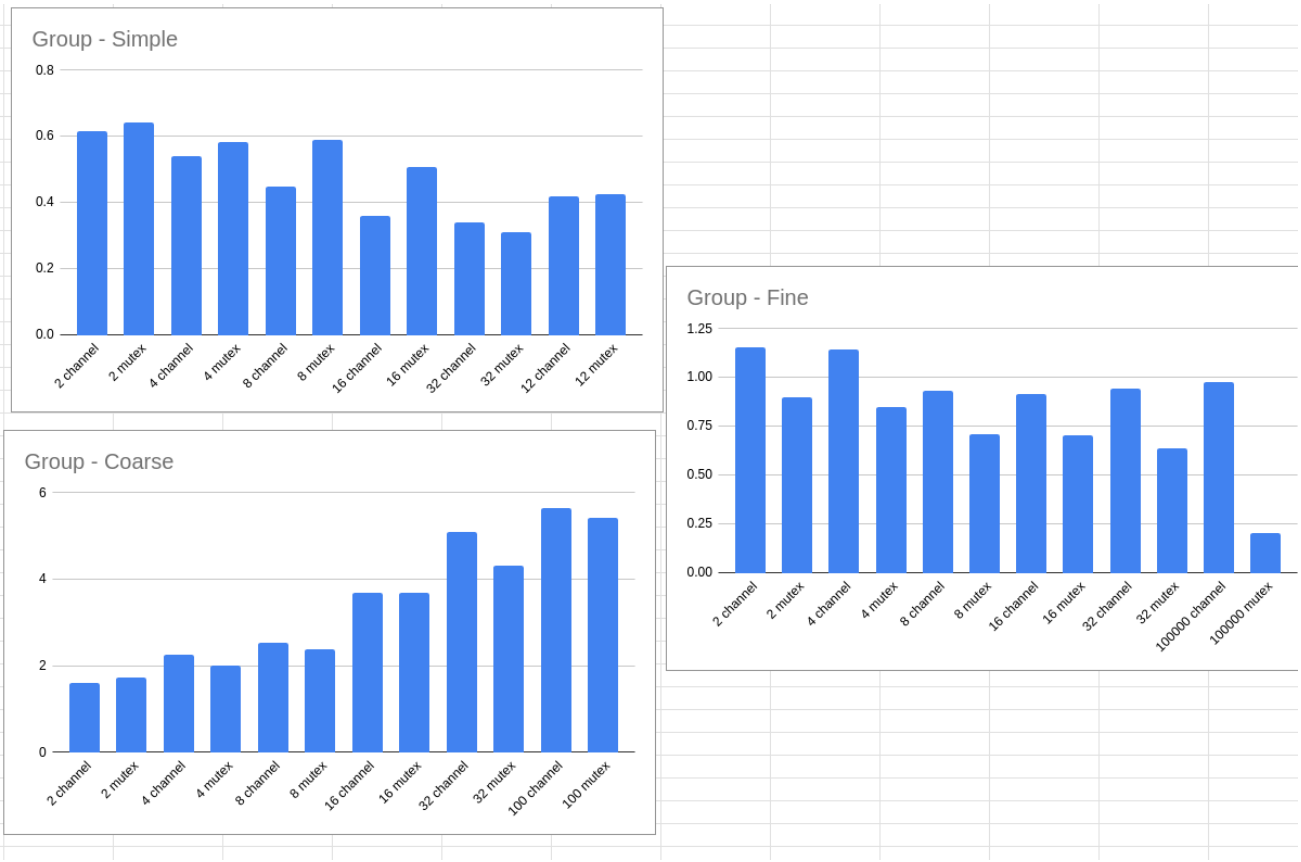


When it comes to the end to end speedup, the results are quite interesting. For the larger two benchmarks, the runtime of the program is mostly dominated by the tree comparisons, so even in the case of the fine benchmark with one goroutine being spawned per tree, there is still an overall speedup in the program execution. As expected, the coarse benchmark and the fine benchmark with a smaller number of worker goroutines see the greatest speedups as they are able to leverage their increased performance in the tree comparisons. For the simple benchmark, even though the overhead of creating the goroutines generally has a higher performance cost than the gain of hashing the trees in parallel, the small number and size of the trees likely causes the cost of accessing and loading the input file to become relatively significant, meaning that the parallel implementation still doesn't see a speedup over the sequential implementation, but the difference in performance is not as large as the difference in just the hashing operation.

Creating BST Hash Groups

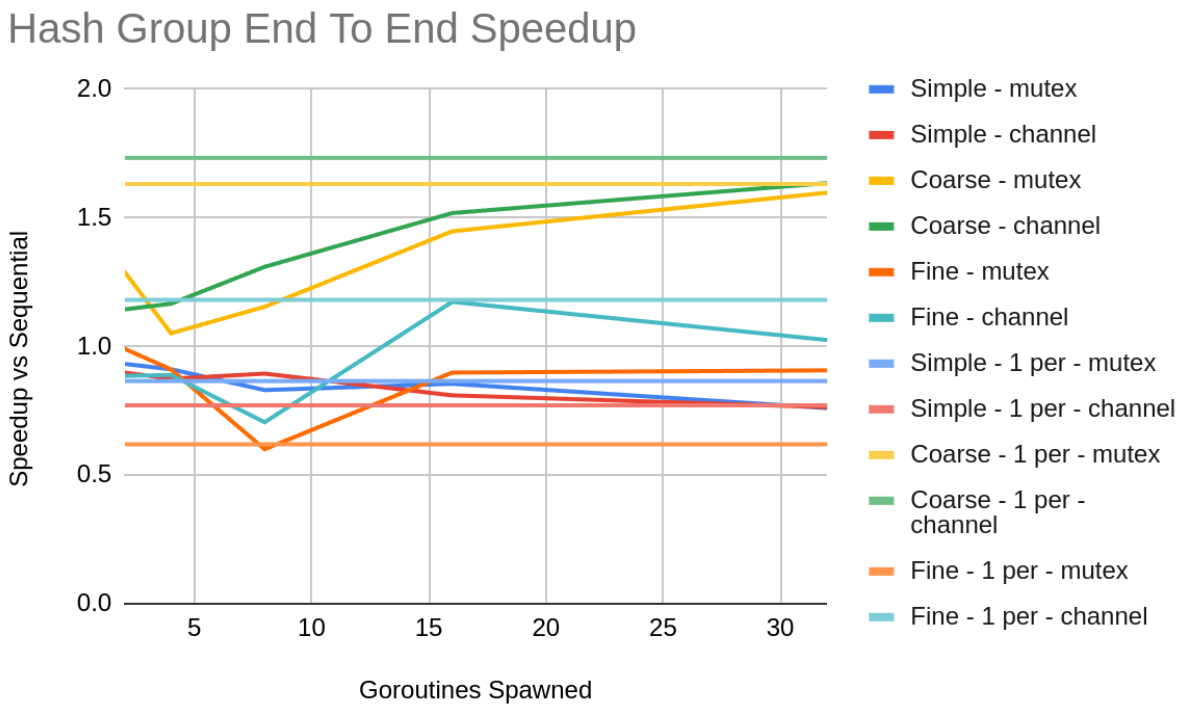


When it comes to creating the hash groups, the coarse benchmark also always sees the greatest speedups, with the one goroutine per tree configuration once again seeing the largest speedup. This is likely due to the same reasons as the hashing, with one hundred goroutines being very manageable to the runtime and the large trees offering the potential to do a lot of somewhat slow work in parallel. The large trees and the time needed to process them is also beneficial because it increases the amount of time that goroutines are not trying to contend for either the mutex or the channel. In the case of the fine benchmark, there are a few configurations where there is a marginal speedup over the sequential implementation, likely once again due to the very large number of trees. However, this large number of trees combined with their small size also means that there is an increased amount of contention for the synchronization primitives used, which reduces the performance increase of the parallel implementation. In the case of the simple benchmark, the small number and size of the trees once again means that the overhead of the goroutines and synchronization primitives generally outweighs the performance gain from parallelism.



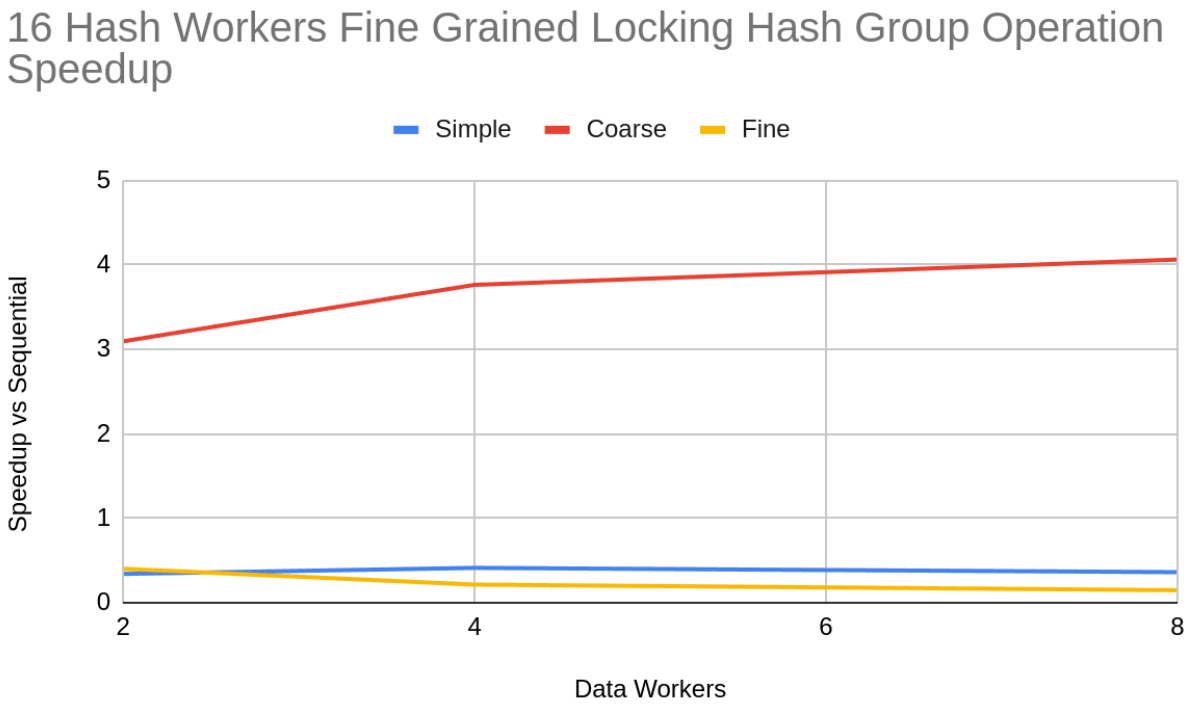
For easier comparison of the channel and mutex configurations, I have created the graph above which compares them side by side for each number of hash workers. Generally, the channel outperforms the mutex, which is to be expected given that the code buffers the channel up to the number of trees. This means that the amount of time that the hash workers spend being blocked is very minimal, as it is just the time that another hash workers is inserting into the buffer which is pretty fast given the small size of the integer pairs, so the hash workers can keep providing the data worker with hashes to place into the map at a very fast rate. Additionally, since the map is only being modified by one goroutine, this means that whatever physical core is running the data worker can keep the hash map in its own cache and continuously update it without needing to invoke any kind of cache coherence protocol for the map. This is in contrast to the mutex, where multiple goroutines can end up fighting for it at once, and the map may also need some cache synchronization to keep its data coherent across the multiple cores that are running the goroutines modifying the map. This evidence is best supported by the results of the comparison of the fine benchmark for one goroutine per tree. In the case of the coarse benchmark, the large trees mean that the goroutines spend a large amount of time doing independent work, generally resulting in contention for the channel and mutex and seeing a speedup for all configurations. In the case of the fine and simple benchmarks, the small tree sizes mean that the goroutines spend

comparatively more time contending for the channel and mutex, resulting in them seeing lesser performance for the work of creating the hash groups.

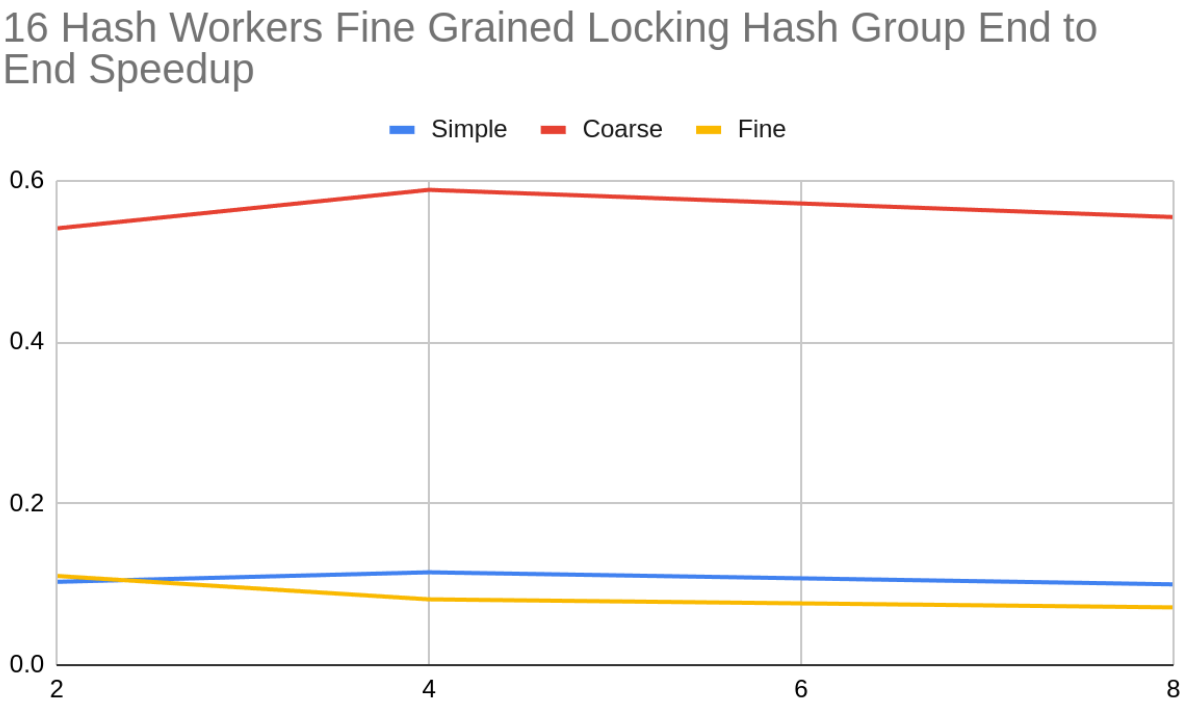


For the overall end to end run time of the hash groups, the coarse benchmark once again sees a speedup pretty much regardless of the configuration due to the large tree comparisons dominating the runtime and making contention for the synchronizing resources have less of a contribution. The earlier mentioned large amounts of contention in the fine and simple benchmarks caused by their small tree sizes causes them to generally have performance similar or slightly worse than the sequential implementation, but the difference in performance is not as drastic as for the grouping alone, likely due to the effects of loading and parsing the file into memory having a relatively meaningful effect on the program's execution time. Overall, while both the mutex approach and the channel approach are equally convenient to implement, the channel approach, provided that the channel is buffered, generally seems to have less of an overhead than having a single mutex to protect the map.

Extra Credit: Fine Grained Locking

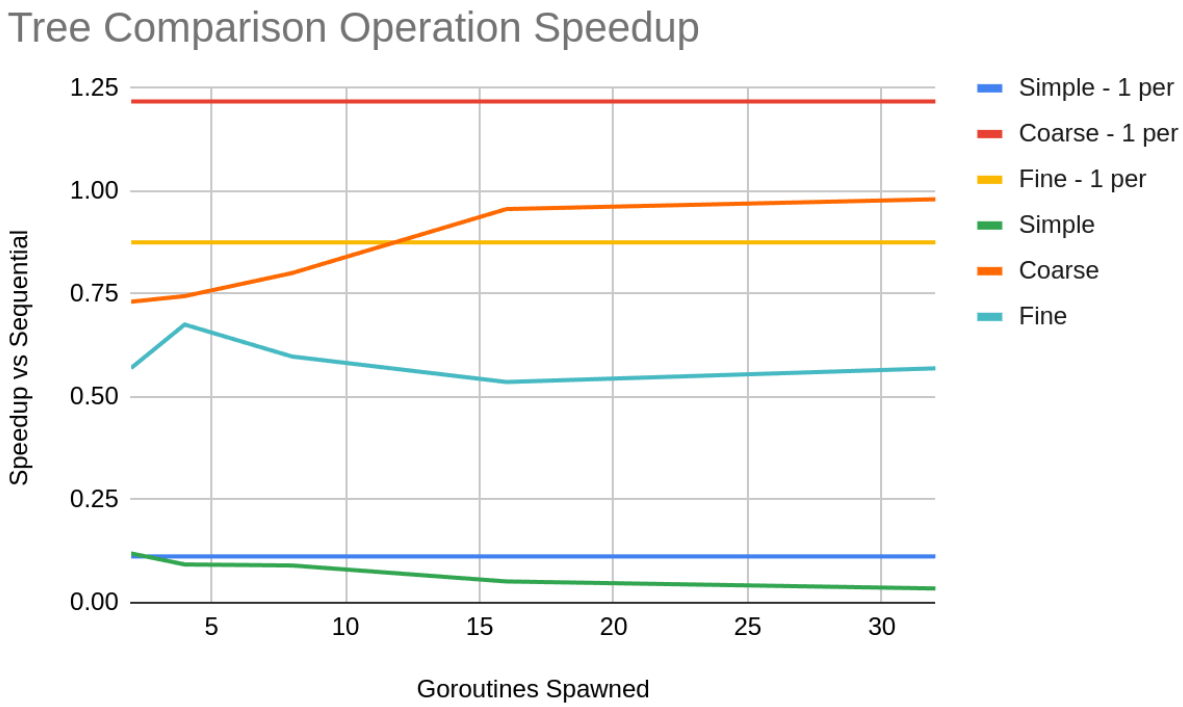


For the fine grained locking approach, the coarse benchmark is the only one which demonstrates a speedup. This is likely once again due to the fact that the tree traversals to compute the hash take longer, reducing the time that goroutines spend contending for the different locks. Furthermore, the number of groups is relatively small and the trees are pretty well distributed amongst those groups, which reduces the number of times that the write lock must be used to add a hash to the map, and the goroutines generally do not contend for the per hash locks that often, further allowing the benchmark to benefit from parallelism compared to the channels and mutexes discussed earlier. This is in contrast to the fine benchmark, which has small trees and many groups. This means that contention for the locks is more frequent, especially for the write mode of the lock which has the potential to turn the program effectively sequential when adding a new hash to the map. The simple benchmark also fails to gain a performance improvement over the sequential implementation in this case, as the overhead of creating all of the locks and goroutines greatly outweighs the benefit of being able to hash and group these pretty small trees in parallel.



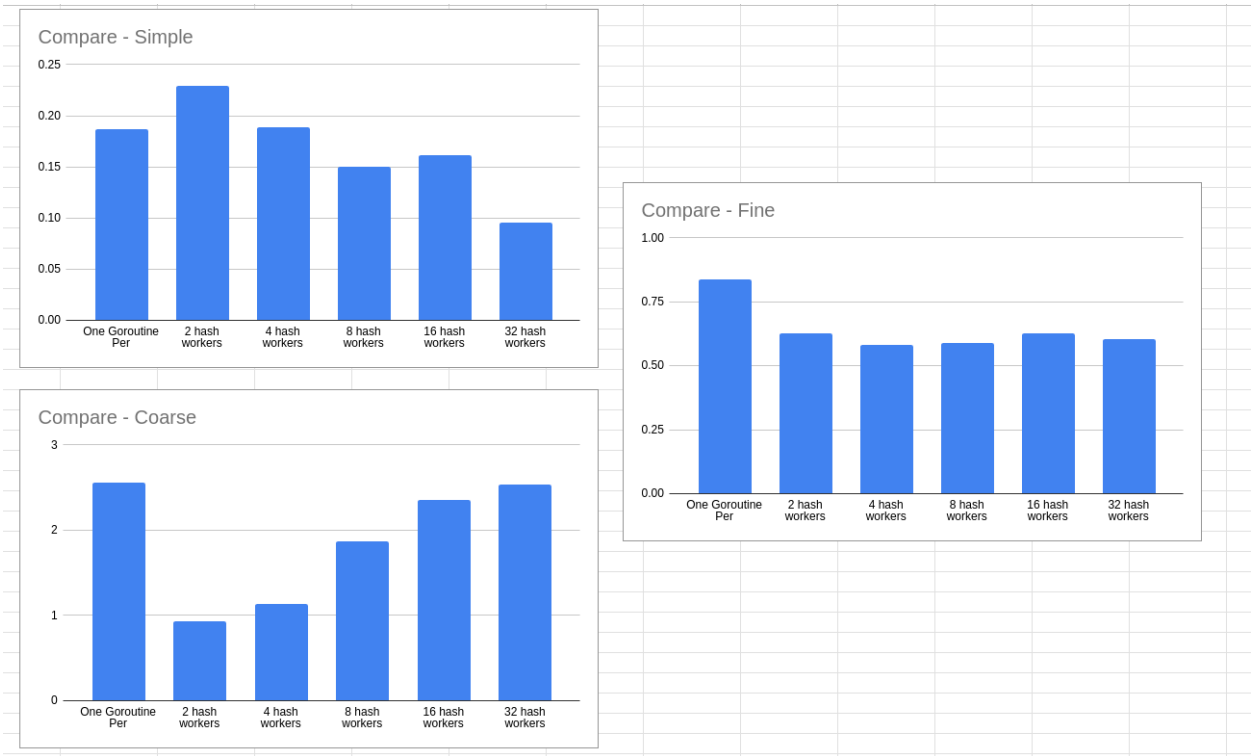
However, when it comes to end to end runtime, the fine grained locking approach never manages to lead to a performance improvement over the sequential implementation. This is likely due to the fact that this approach requires the creation of many more synchronization primitives, which are generally heavier objects that require more involvement from the runtime, reducing the overall end to end speedup. For the fine and simple benchmarks, this cost combined with the slower performance of this implementation compared to the sequential implementation explains the lack of a speedup in end to end time compared to the sequential implementation. For the coarse benchmark, while there may be a speedup for the grouping portion, in the bigger picture the costs of all of the created synchronization variables are likely the cause of the overall program not seeing a speedup over the sequential implementation.

Comparing Trees

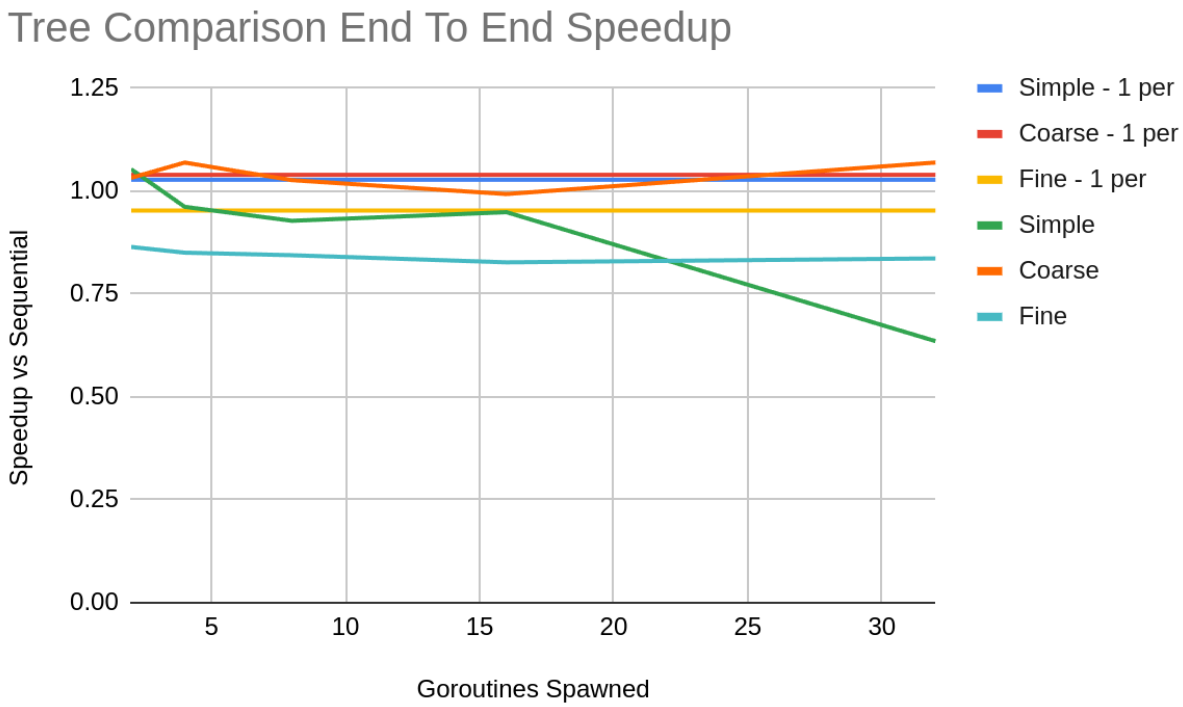


When it comes to comparing the tree comparison operation speedups, it is important to recall that the step of creating the hash groups also stores the in order tree traversals inside of a slice of slices, meaning that comparing the trees can actually be done very quickly, as the comparison operation is just comparing two slices. In the binary tree representation, the nodes may be all over memory, causing many cache misses and exhibiting behavior that is relatively difficult for the data prefetcher to account for. Meanwhile, the slices are all contiguous in memory, leading to them having great cache efficiency and being very easy to prefetch. Furthermore, no auxiliary data structures need to be made as there is no need to perform the in order traversal. All of these factors explain why the general trend in the tree comparison data is that almost no configuration leads to a speed up over the sequential algorithm. The one exception to this rule is the coarse benchmark, which sees a slight speedup which is likely due to the fact that the tree comparisons take longer due to the larger tree sizes, and due to the fact that spawning a goroutine per tree has no contention for synchronization primitives aside from the barrier which synchronizes the goroutines at the end. Another important trend in this data is that spawning a goroutine per comparison is generally better for performance than using the custom buffer. This can be attributed to the one goroutine per tree has no contention for synchronization primitives the same way that the concurrent buffer does. The concurrent buffer is composed of a mutex and two condition variables, so even though placing and taking from the buffer is relatively fast,

there is still contention among the goroutines for these primitives which can reduce the performance of the program.



Out of curiosity, I also removed the storing of traversals in the grouping stage to see what the effect on the speedup of the operation would be. As expected, the coarse benchmark is now able to see a speedup regardless of the configuration as the longer time to traverse the tree means allows for more parallel work, and in the case of the concurrent buffer this means that there is even smaller portion of time that the goroutines are contending for the buffer. In the case of the simple and fine test cases however, the small tree sizes mean that the goroutines still spend a decent portion of their time contending for the concurrent buffer, resulting in there not being a speedup over the sequential implementation. This seems to indicate that managing a thread pool in a manner similar to what the concurrent buffer is doing could be worthwhile, provided that the work is not a bunch of small tasks that get quickly completed and then cause the goroutines to contend for the buffer distributing the work.



For the end to end speedup, most of the configurations are roughly similar to the sequential implementation, but the general trend is that these configurations often perform worse than the sequential implementation, with any provided speedups being fairly marginal. Granted, with this data is is important to note the the preceding hashing and grouping step was executed sequentially for all configurations to provide a the most fair baseline for all of these configurations, but combined with the fact that the comparison step is greatly simplified by preprocessing done during the grouping stage means that the potential for the parallel implementation to show a speedup is fairly limited. The one goroutine per comparison configurations are generally closer than the concurrent buffer configurations, likely once again due to them not having any contention for synchronization primitives as mentioned earlier. Furthermore, the fine benchmark and the simple benchmark with the concurrent buffer perform particularly poorly, likely due to their small trees leading to very fast tree comparisons and more frequent contention for the concurrent buffer. In the case of the fine benchmark with one goroutine per tree, the reduction in performance likely once again comes from the fact that there is a large number of comparisons which need to be made, meaning that the cost of creating all of the goroutines begins to become a more significant contributor to the execution time.