# Lab 3 Report

## Step 1: Create a sequential solution

|  |  |  |
| --- | --- | --- |
| **Time Taken to Run Sequential Algorithms** | | |
|  | **coarse.txt** | **fine.txt** |
| **hashTime** | 0.02594934 | 0.031723284 |
| **hashGroupTime** | 0.02594934 | 0.031723284 |
| **compareTime** | 0.953952061 | 2.866784366 |

## Step 2: Parallelize hash operations

### First Implementation

This should be done by spawning a goroutine for each BST.

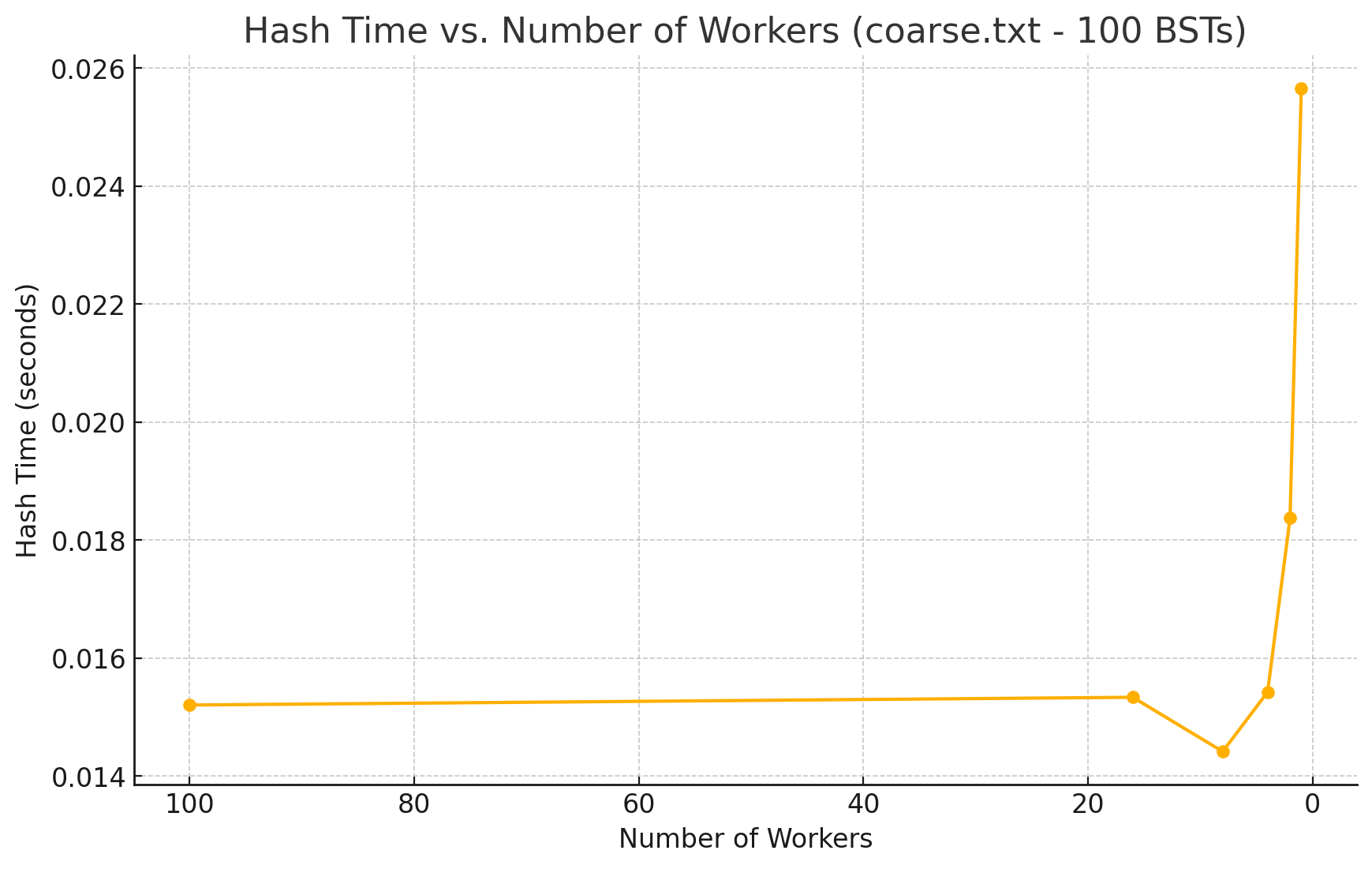
|  |  |  |
| --- | --- | --- |
|  | **coarse.txt (100 BSTs)** | **fine.txt (100000 BSTs)** |
| **hashTime in seconds** | 0.0154681 | 0.036130771 |

### Second Implementation

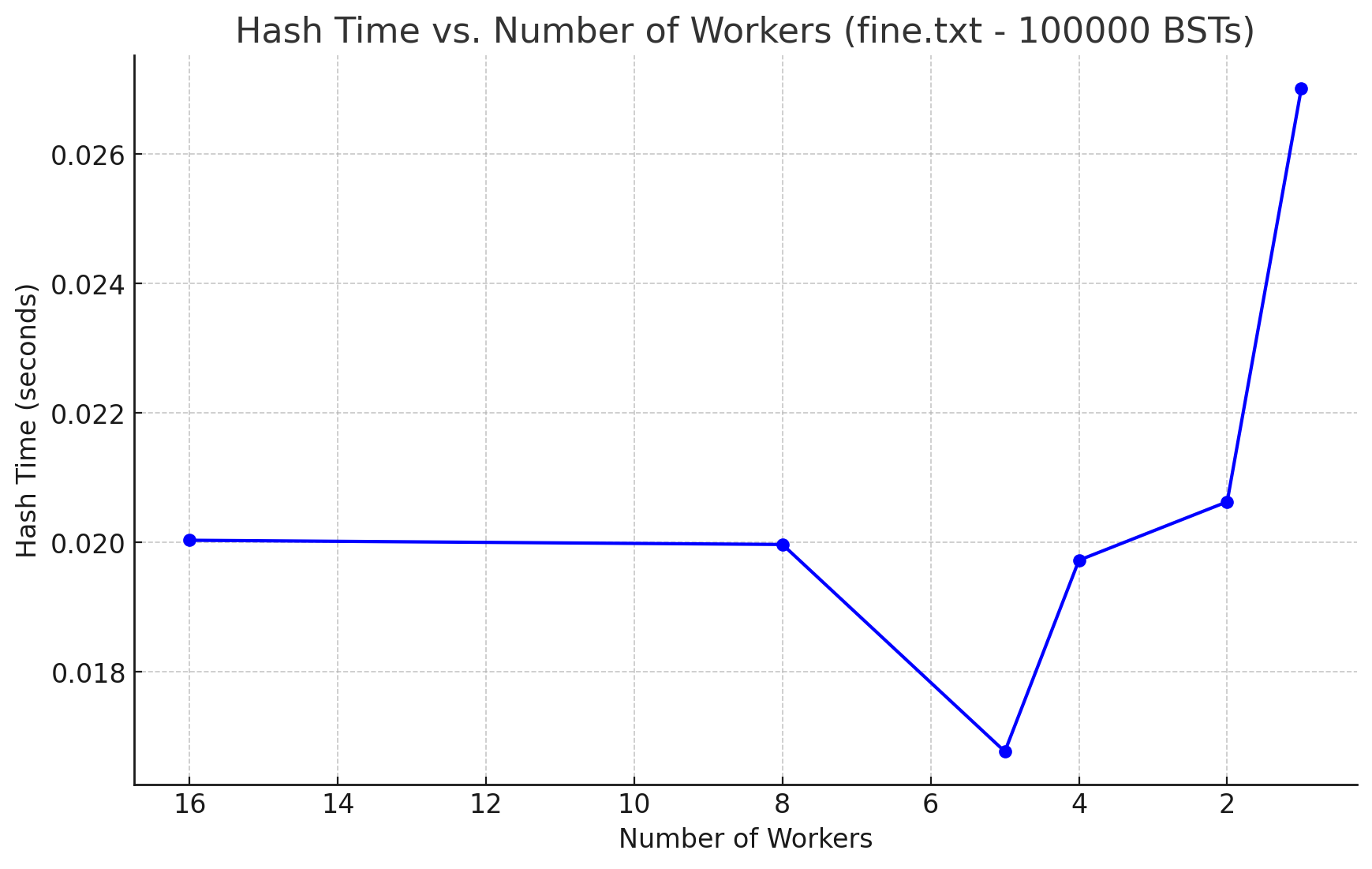
Spawning hash-workers goroutines (threads) and having them iterate over the available BSTs.

Running code with several different values of hash-workers on both coarse.txt and fine.txt.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **coarse.txt (100 BSTs)** | | | | | | |
|  | **workers = 100** | **workers=16** | **workers=8** | **workers=4** | **workers=2** | **workers=1** |
| **hashTime in seconds** | 0.015205819 | 0.015336481 | 0.014417725 | 0.015425107 | 0.018374174 | 0.025650518 |



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **fine.txt (100000 BSTs)** | | | | | | | |
|  | **workers = 100000** | **workers=16** | **workers=8** | **workers=5** | **workers=4** | **workers=2** | **workers=1** |
| **hashTime in seconds** | 0.127373686 | 0.02003266 | 0.0199675 | 0.0167678 | 0.01972406 | 0.02062743 | 0.027013336 |



#### Which implementation is faster (and by how much)?

My expectation is that the second approach should be faster for a specific number of workers that balance between parallelism and threading overhead.  
For the coarse file, there is no considerably faster implementation. Both are nearly the same speed; However, the first implementation is easier and more straightforward.

While for the fine.txt file, the first approach is clearly slower than the second approach. A weird finding is that the sequential approach is faster than the first parallel implementation.

#### Can Go manage goroutines well enough that you don't have to worry about how many threads to spawn anymore?

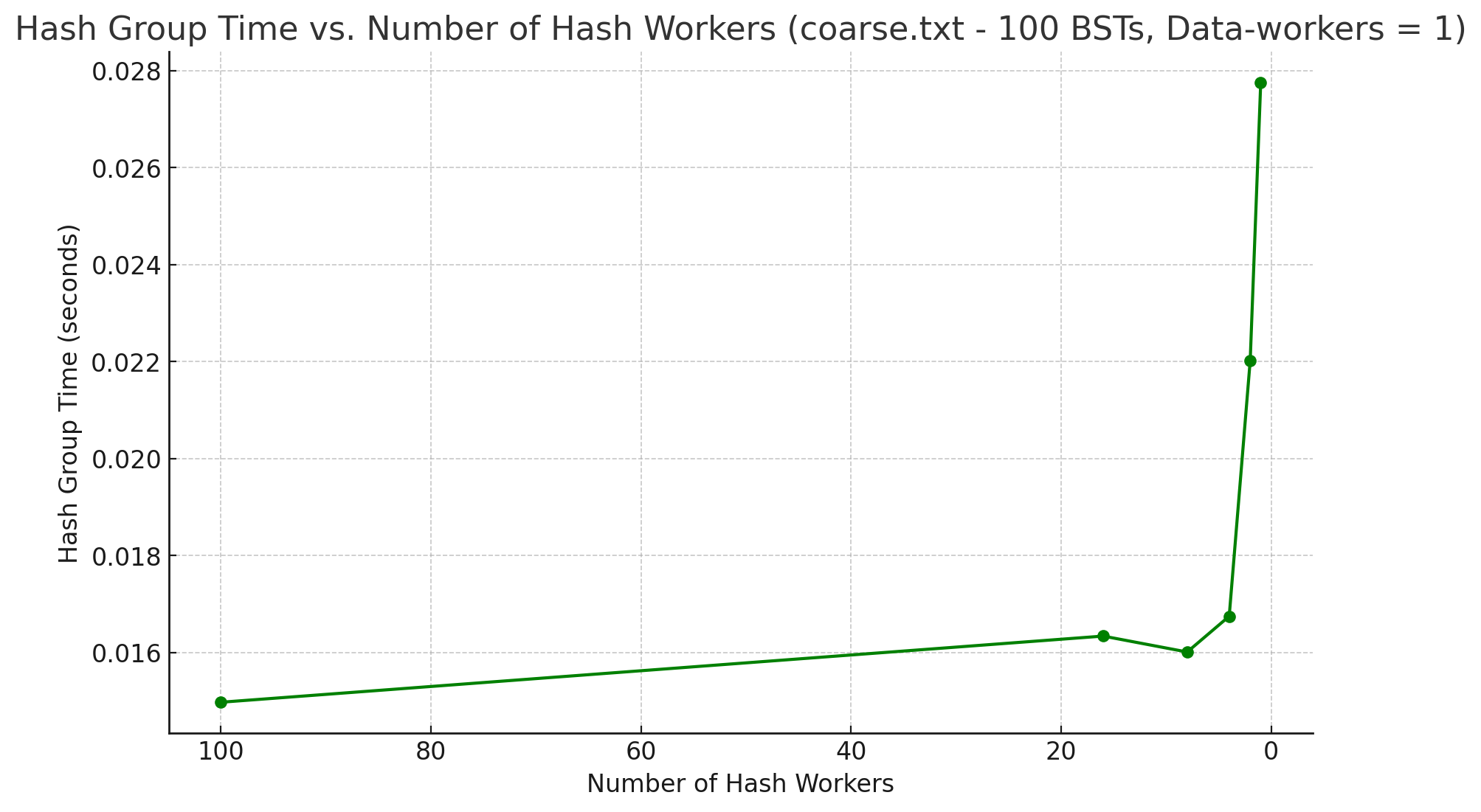
No, I don't think so. Since, it will always depend on the problem we are trying to solve.

## Step 2: Hash grouping operations

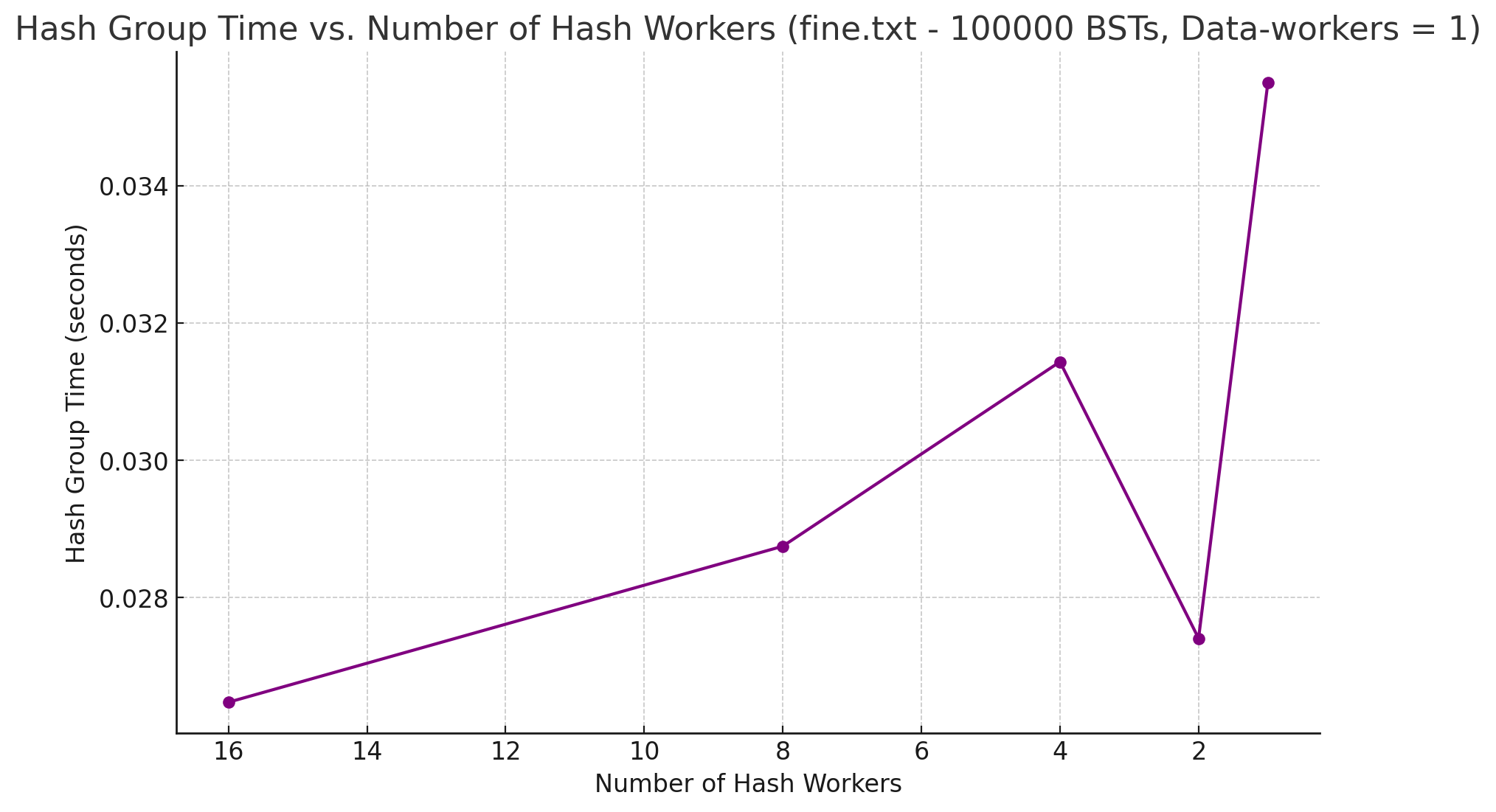
### First Implementation

Single thread to add the hashes to the map and then group the BSTs indices (sequential grouping)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **coarse.txt (100 BSTs) (Data-workers = 1)** | | | | | | |
|  | **hash workers = 100** | **hash workers = 16** | **hash workers = 8** | **hash workers = 4** | **hash workers = 2** | **hash workers = 1** |
| **hashGroupTime in seconds** | 0.01497234 | 0.016337 | 0.016008588 | 0.01674644 | 0.02202435 | 0.02775097 |



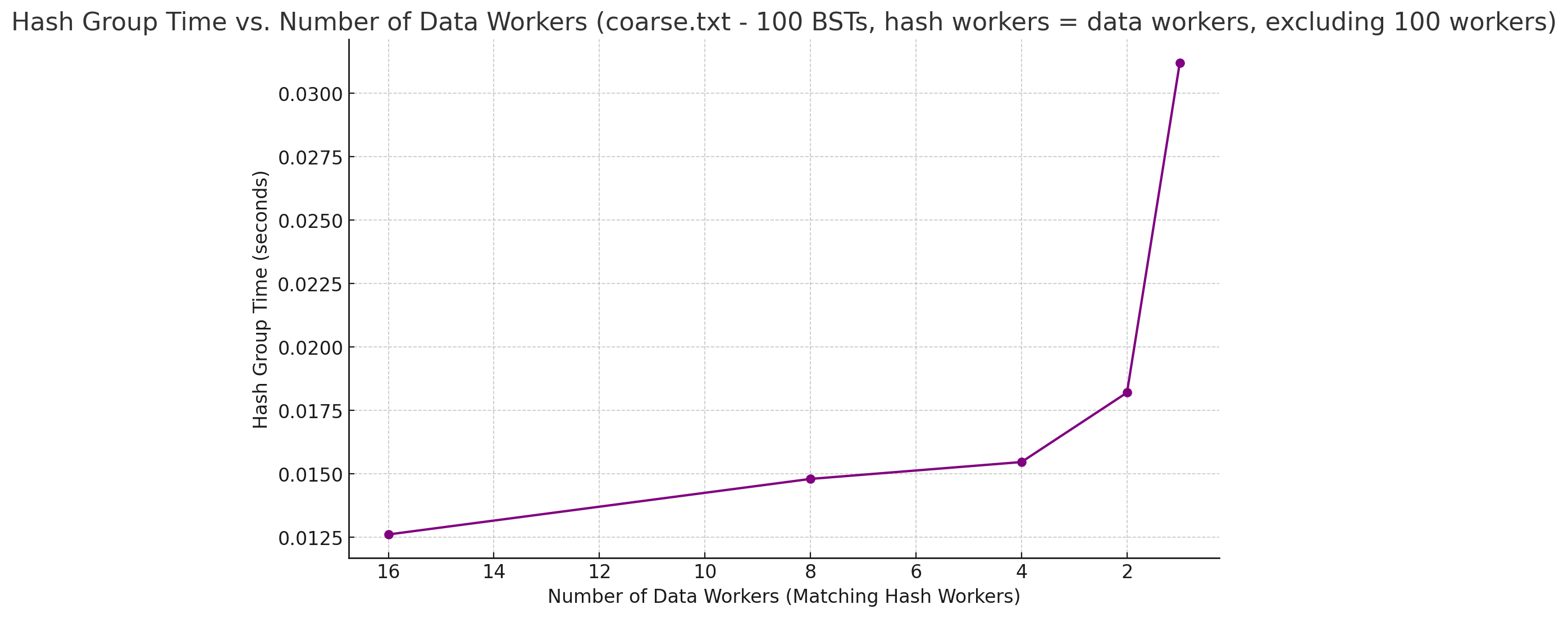
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **fine.txt (100000 BSTs) (Data-workers = 1)** | | | | | | |
|  | **hash workers = 100000** | **hash workers = 16** | **hash workers = 8** | **hash workers = 4** | **hash workers = 2** | **hash workers = 1** |
| **hashGroupTime in seconds** | 0.150663617 | 0.026476157 | 0.028748095 | 0.031433987 | 0.027406791 | 0.03549789 |



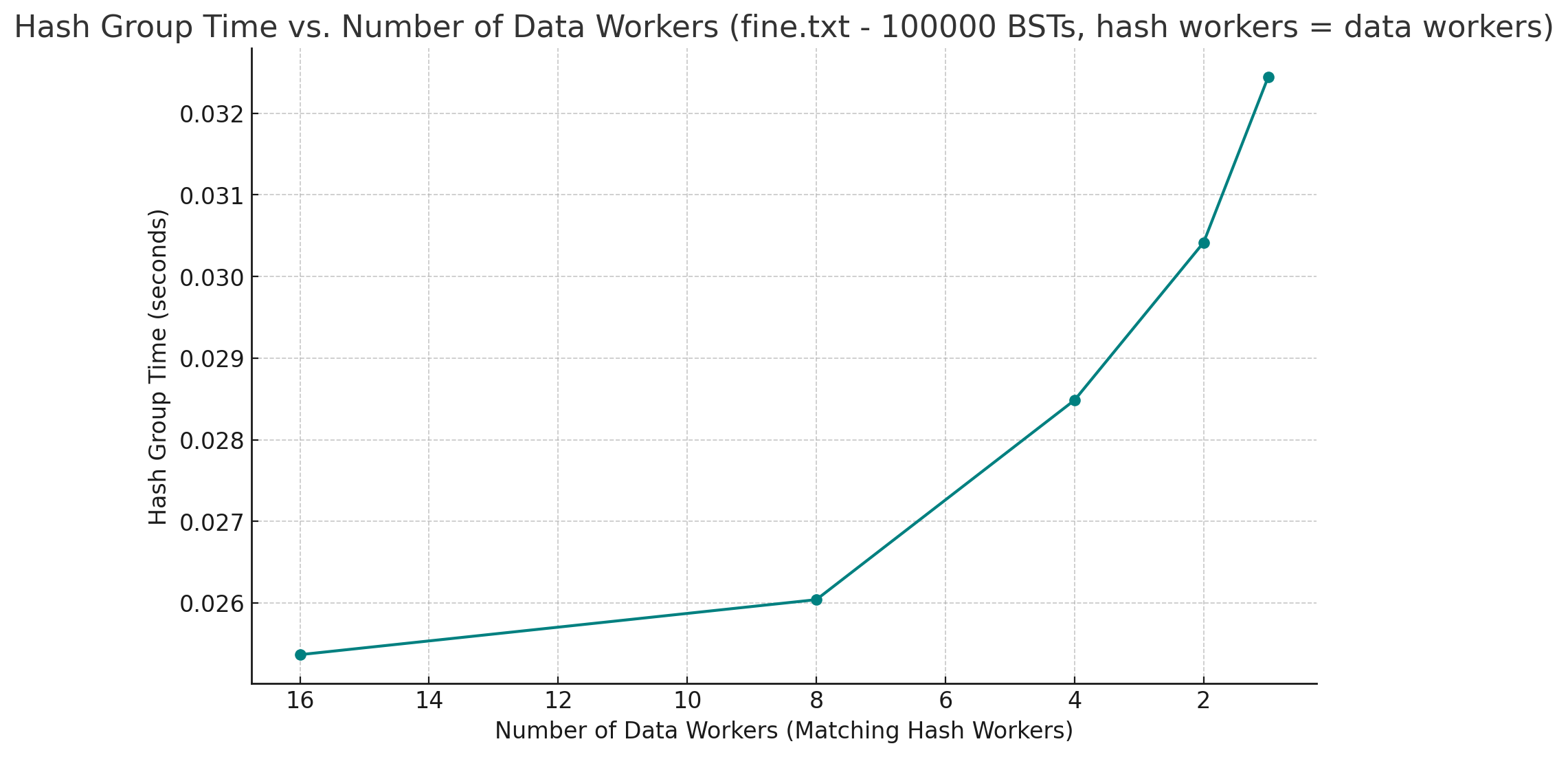
### Second Implementation

A single lock protects the data structure, and each thread must acquire the lock to add its result to the data structure (parallel grouping)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **coarse.txt (100 BSTs)** | | | | | | |
|  | **Data-workers = 100** | **Data-workers = 16** | **Data-** **workers =**  **8** | **Data-workers = 4** | **Data-workers =**  **2** | **Data-workers =**  **1** |
| **hashGroupTime in seconds**  **(hash workers = data workers)** | 0.124262424 | 0.01261 | 0.014799277 | 0.015464723 | 0.018200979 | 0.031195466 |



|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **fine.txt (100000 BSTs)** | | | | | | |
|  | **Data-workers = 100000** | **Data-workers = 16** | **Data-workers =**  **8** | **Data-workers =**  **4** | **Data-workers =**  **2** | **Data-workers =**  **1** |
| **hashGroupTime in seconds (hash workers = data workers)** | 0.15593989 | 0.0253668 | 0.0260407 | 0.0284846 | 0.03041615 | 0.032442327 |



#### Compare the speed of both implementations.

My expectation is that the second approach should be faster for a specific number of workers that balance between parallelism and threading overhead.

##### coarse.txt (100 BSTs)

* **First Implementation**:
  + hashGroupTime improves with an increasing number of hash workers, indicating better performance as tasks are distributed among more workers.
  + **Best time** with 100 hash workers: **0.0149734 seconds**.
* **Second Implementation**:
  + hashGroupTime also improves as more data workers are added, but the times are generally similar to the first implementation at higher worker counts.
  + **Best time** with 16 data workers: **0.01261 seconds**, slightly faster than the first implementation.
* **Conclusion**: In this case, the second implementation is marginally faster, especially with a high number of data workers, likely due to parallel execution compensating for lock contention.

##### fine.txt (10000 BSTs)

* **First Implementation**:
  + hashGroupTime improves with more hash workers, with the best time at 16 hash workers: **0.026476157 seconds**.
  + Performance generally decreases slightly as hash workers are reduced, showing a dependence on a high number of workers for optimal performance.
* **Second Implementation**:
  + hashGroupTime shows similar improvement as data workers increase, but times are generally faster than the first implementation.
  + **Best time** with 16 data workers: **0.0253668 seconds**, slightly slower than the first implementation in this case.
* **Conclusion**: The second implementation is slightly faster for fine.txt when dealing with a moderate number of hash workers, while the second implementation falls behind by a small margin at the same number of workers.

The second approach adds more overhead due to lock contention. Especially at high concurrency, where multiple goroutines frequently compete for the lock. This overhead can result in the first implementation being faster in specific cases, particularly with extremely large datasets and high worker counts.

I find the second approach to be simpler since we don't need a channel to send data to the main thread to add data into the data structure we are working on.

##### How much faster are they compared to a single thread?

for coarse.txt, the firstimplementation is 2.16 faster than a single thread. The second implementation is 2.573 faster.

for fine.txt, the first implementation is 1.34 faster, the second is 1.4 faster.

## Step 3: Parallelize tree comparisons

### First Implementation

spawn a goroutine to do each comparison. This approach is straightforward but may lead to excessive overhead with very high concurrency due to goroutine management.

|  |  |  |
| --- | --- | --- |
|  | **coarse.txt (100 BSTs)** | **fine.txt (100000 BSTs)** |
| **compareTreeTime in seconds** | 0.63918594 | 5.12444431 |

### Second Implementation

spawn comp-workers threads to do the comparisons and use a concurrent buffer to communicate with them. This effectively creates a thread pool where threads repeatedly pick up tasks from the buffer and process them. It adds some complexity due to buffer management and thread synchronization but limits the number of active threads at any time.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **coarse.txt (100 BSTs)** | | | | | | | |
| **Comp workers** | **1** | **2** | **4** | **8** | **16** | **32** | **64** |
| **compare** **Time in seconds** | 1.001 | 0.770 | 0.666 | 0.597 | 0.577 | 0.559 | 0.552 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **fine.txt (100000 BSTs)** | | | | | | | | |
| **Comp workers** | **1** | **2** | **4** | **8** | **16** | **32** | **64** | **128** |
| **compareTime in seconds** | 3.455 | 3.265 | 3.065 | 5.408 | 3.089 | 3.191 | 3.410 | 3.716 |

#### Analysis of Performance

My expectation is that the second approach should be faster for a specific number of workers that balance between parallelism and threading overhead.

##### coarse.txt (100 BSTs)

* First Implementation:
  + compareTreeTime: 0.63918594 seconds

Spawning a goroutine per comparison worked well here, as the total time for 100 BST

comparisons is quite low compared to using a single thread (which would take much longer).

* Second Implementation (with comp-workers):

The performance improves with the increase in comp-workers up to 64 workers.

Best time: **0.552 seconds with 64 workers**, which is faster than the First Implementation's time. The results show diminishing returns after 16 workers, suggesting that increasing workers beyond a certain point adds more overhead without substantial improvement.

#### **fine.txt (100,000 BSTs)**

* **First Implementation**:
  + **compareTreeTime**: **5.12444431 seconds**
  + Spawning a goroutine for each comparison incurs considerable overhead with a very high number of comparisons (100,000), making this approach slower for large datasets.
* **Second Implementation** (with comp-workers):
  + The best performance is at 4 workers: **3.065 seconds**.
  + Interestingly, performance decreases with higher worker counts, especially at 8 workers (5.408 seconds) and beyond.
  + This suggests that, for this dataset, 4 workers strike a balance between parallelism and managing the overhead of synchronization with the concurrent buffer.

##### Is the additional complexity of managing a thread pool worthwhile?

* Yes, for large datasets (e.g., fine.txt), the Second Implementation’s controlled concurrency provides significant performance benefits.
* For smaller datasets (e.g., coarse.txt), both implementations perform comparably, but the Second Implementation still shows a slight advantage by maintaining predictable resource usage.