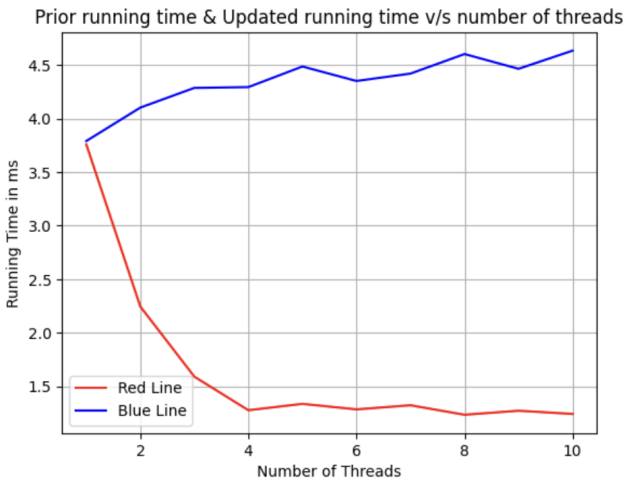
*What circumstances cause an entry to get lost?*

In this code, entries are lost when a thread wants to retrieve a key from the hash table but it's missing. This happens due to a lack of synchronization between the threads. Looking at the put_phase and get_phase functions, we can see that they aren't using any locks or synchronization mechanisms. In the put_phase function, when multiple threads want to insert keys into the hash table, they might try to insert keys into the same bucket, thus causing overwriting. Also, in the get_phase function, multiple threads might try retrieving keys simultaneously, and there aren't any locks to handle who gets to access which bucket when, so the threads might interfere with each another's activities and thus cause keys to be incorrectly marked as 'lost'.

In this part, we modified the code by adding a global lock for the insert operation, meaning that every thread that wants to insert into the table will have to acquire a lock, do the insertion and then release the lock. We also ensured that while retrieving the key from the hashtable, we are also locking the resources to prevent multiple simultaneous access. We made use of the *pthread_mutex* library for this locking mechanism.

GRAPH: *Prior running time v/s updated running time*




*The time overhead for the code with mutex locks is ~162%. This means that the code with mutex locks is 1.62 times slower than the original parallel_hashtable code without locking, so this much slow down is required to guarantee correctness in the code (no loss of any keys).*

*How we came up with this estimate:*

*Overhead = (average running time (original code) - average running time (mutex code) / average running time(original code)) *100*

*Average running time(original code)=*
*3.758267+2.244175+1.589807+1.276486+1.335996+1.284497+1.322740+1.234120 +1.271743+1.241847/10*
*=1.6559678*
*Average running time(mutex code)*
*=3.790491+4.102659+4.286679+4.294018+4.486191+4.350985+4.420049+4.602597+4.464727+4.634200/10*
*=4.343259600000001*

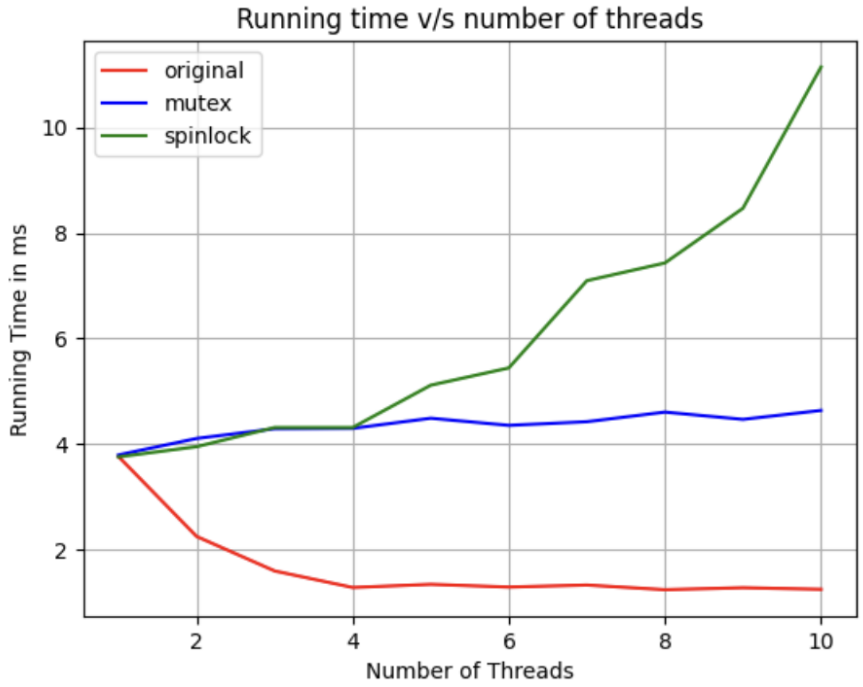
*Overhead*
*=4.343259600000001-1.6559678/1.6559678 *100*
*=162.27923030870534*

*Therefore, a ~162% increase in time for the mutex code.*

## PART 3 - Spinlocks

*If you were to replace all mutexes with spinlocks, what do you think will happen to the running time? Write a short answer describing what you expect to happen, and why the differences in mutex vs. spinlock implementations lead you to that conclusion*

- If we replace all mutexes with spinlocks, the runtime should logically increase. We know that mutexes put the calling thread to sleep (after a few failed acquisitions) if it is already held, thus decreasing the busy waiting time and reducing CPU usage while we wait for the lock to be released.
- Spinlocks are implemented such that the thread that fails to acquire the lock continually retries (spinning) to get it. While this wastes CPU time, it can lead to better performance.
- In this situation there are a large number of threads contending for locks in the hash table with shorter critical sections. Intuitively, due to constant spinning, the running time will increase for spinlocks.




*The time overhead for the code with spinlocks over the original/unsafe code is ~268%. This means that the code with mutex locks has a 268% slow down than the original parallel_hashtable code without locking.*

*The time overhead for the code with spinlocks over the mutex-based code is ~40%. This means that the code with spin locks has a 40% slow down than the original parallel_mutex code with mutex locks as the thread safety mechanism.*

*Average running time(original code)=*
*=(3.758267+2.244175+1.589807+1.276486+1.335996+1.284497+1.322740+1.234120*
*+1.271743+1.241847)/10*
*=1.6559678*
*Average running time(mutex code)*
*=(3.790491+4.102659+4.286679+4.294018+4.486191+4.350985+4.420049+4.602597+4.464727+4.634200)/*
*10*
*=4.343259600000001*
*Average running time(spinlock code)*
*=(3.753446+3.946501+4.310338+4.310245+5.112699+5.439992+7.095014+7.430444+8.468255+11.145421)*
*/10*
*=6.1012355*
*Overhead(spinlock v/s original code)*
*=(6.1012355-1.6559678/1.6559678)*100*
*=268.4392595073406%*
*Overhead(spinlock v/s mutex code)*
*=(6.1012355-4.34325960000001/4.34325960000001)*100*
*= 40.47595727411729%*

## Part 4 - Mutex, Retrieve Parallelization

*When we retrieve an item from the hash table, do we need a lock?*

We do not need keys for general retrieval operations.
These are typically read-only operations and therefore it does not matter if multiple threads try to access the hash table and, more precisely, scan the buckets for the required keys.
Even if two threads want to access the same lock, since it is read-only we don't need the lock when searching for the required group entry.
This achieves parallelism because threads do not wait for other threads to release their locks on the pool before they can read their own locks.

So, we modify the *parallel_mutex_opt.c* by removing the locks from the "retrieve" function, hence achieving parallelization (threads don't wait for the locks held by other threads to be released) of the retrieve operation.

This modified fetch function allows multiple threads to simultaneously access and look up keys in the hash table without the need for any locking mechanism, allowing efficient parallelization of read-only operations.

## Part 5 - Mutex, Insert Parallelization

*Describe a situation in which multiple insertions could happen safely*

When threads are writing to different buckets, we do not require locks. In this scenario, multiple threads could easily be writing to the hashtable (but to different buckets) and that way, the parallel insert operations would also be safe.

In the code, we'll modify the code to handle the situation wherein two threads try writing to the same bucket using a lock, so we don't need a global lock anymore. Hence, when different threads try writing to different buckets, they'll be able to do so in parallel without a global lock, but a lock on the bucket it needs to write to instead.

In simple words, this modification introduces a per-bucket lock instead of a global lock. This will ensure that many threads can enter their respective buckets, but the locking and unlocking only happens at the particular bucket, so that multiple threads can't write into the same bucket at the same time. This also decreases the size of the critical section.