Project 2

CS342 - Operating Systems

Efe Acer - 21602217
Yusuf Dalva - 21602867

Bilkent University, CS

Contents

Introduction

The term *Thread* refers to the thread of execution of a computer program. Threads are used to split the execution of a program into multiple *concurrently* running tasks. Hence, they can significantly speed up the execution of a program, especially when the program is sufficiently parallelizable. The performance gain that a set of threads can introduce is given formally by:

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}} \tag{1}$$

, where S denotes the non-parallelizable portion of a program and N is the number of threads. The right hand side of (1) gives an upper-bound for the potential performance gain threads can introduce, which can be very large when S is small and N is large. This formula is known as *Amdahl's Law* [1].

Threads have inarguable advantages, however one must use them with care.

The biggest issue that comes into play is *synchronization*. Threads are usually allowed to access and manipulate shared data. Unfortunately, concurrent access to shared data may result in inconsistent behavior. Synchronization, in our case, refers to the orderly execution of threads to prevent data inconsistencies [2].

Operating systems typically provide routines to create threads and tools to ensure synchronization. A library that can be used for these purposes is `pthreads`.

The rest of this report will revolve around threads and their synchronization. The next section provides a description of an experiment we have conducted in order to investigate the effects of threads and synchronization tools on a program's performance.

Experiment

Our experiment is dependent on a thread safe hash table that we have implemented using `pthread_mutex` locks. A mutex lock is a relatively old tool to ensure mutually exclusive access to the shared data. The hash table makes use of mutex locks to guarantee that no two threads can insert, update, retrieve or delete an item in a way that may result in an inconsistent hash table state.

In the experiment, we perform a total of W operations for each operation type. We use T number of threads, thus each thread performs W/T operations for each type. The hash table has N buckets and it uses K locks to protect its buckets. A lock protects a certain region of buckets and there are N/K, M regions in total.

We are interested in the effect of K and T on the performance. We do this by measuring the time elapsed from the initialization of the hash table to the deallocation of it after W operations from each type are successfully performed. First we keep N, W and K constant and change T to observe its effect on performance, then repeat the same procedure keeping T constant instead of K.

For the first part of the experiment where we change T, N is taken as 100 and K is taken as 10. For the other part, N is taken as 1000 and T is taken as 10. W is taken as 100000 for both parts. The results of the experiment are given in tables and plots in the next section.

Data and Interpretation

The table below displays execution times for different values of T:

T	N	K	W	CPU Time
1	100	10	100000	**2.84**
2	100	10	100000	**4.02**
4	100	10	100000	**3.46**
5	100	10	100000	**3.64**
10	100	10	100000	**3.42**
20	100	10	100000	**3.65**
25	100	10	100000	**3.71**
40	100	10	100000	**3.81**
50	100	10	100000	**4.16**
80	100	10	100000	**4.76**
100	100	10	100000	**4.32**
500	100	10	100000	**5.85**
1000	100	10	100000	**7.89**

Figure 1: Table for T vs CPU time

The values in the table are given as a plot in the following figure (note that T=500 and T=1000 are excluded):

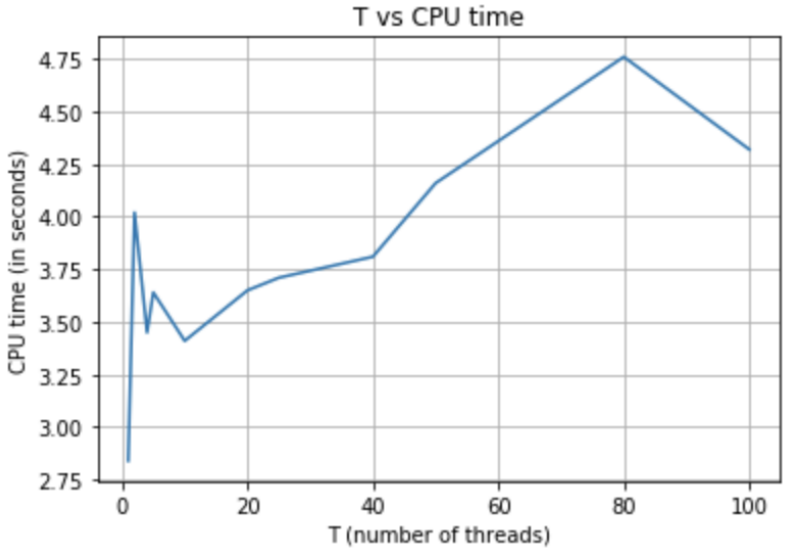

Figure 2: Plot for T vs CPU time

The table below displays execution times for different values of K:

T	N	K	W	CPU Time
10	1000	**1**	100000	**2.66**
10	1000	**5**	100000	**1.22**
10	1000	**10**	100000	**0.85**
10	1000	**20**	100000	**0.76**
10	1000	**40**	100000	**0.55**
10	1000	**50**	100000	**0.50**
10	1000	**100**	100000	**0.39**

Figure 3: Table for K vs CPU time

The values in the table are given as a plot in the following figure:

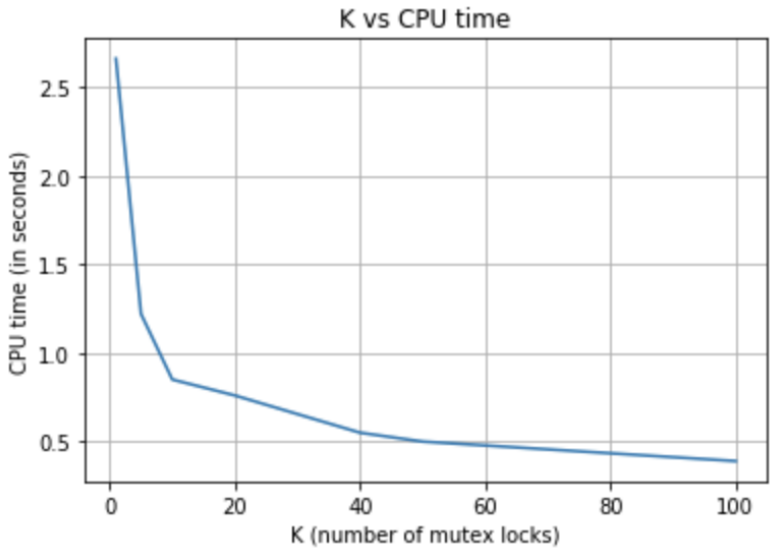

Figure 4: Plot for K vs CPU time

As it can be seen from Figure 2, number of threads T does not have a strong correlation with the CPU time. In fact, the program executes fastest when a single thread is used. There are several reasons for this.

One is that the computer architecture in which we have conducted this experiment is quad-core, meaning that after 4 threads we do not have complete parallelism but instead we have concurrency. Concurrent execution is expected to increase performance, however for such a small scale program context switch times themselves introduce a significant delay. This also explains why we observe a general increase in the CPU time as T increases.

Another reason is that for a fixed value of K, it is certainly the case that the threads will wait for each other when there are more than K threads. K gives the total number of mutex locks as we have mentioned earlier, in this case we have 10 locks. Since each lock is used to allocate a region of the hash table to a single thread, we can at most have 10 threads working on different regions. 11th thread necessarily waits for one of the working threads to finish its execution. This implies that we must increase K to observe true concurrency.

It is obvious from Figure 4 that K and CPU time have a strong negative correlation. Therefore, K is positively correlated with CPU performance. In fact, we observe an exponential decay in CPU time as K increases.

This is an expected result since the increase in K allows more resources to be allocated to different threads. Hence, K threads can execute concurrently. However, as K increases CPU time starts to decrease slower, meaning the first derivative of CPU time is positively correlated with K. This is also meaningful, since:

$$\frac{d(speedup)}{dN} = \frac{1-S}{(1+(N-1)S)^2} \tag{2}$$

Observe that:

$$\lim_{N\to\infty} \frac{1-S}{(1+(N-1)S)^2} = 0 \tag{3}$$

which implies that as the number of threads that can execute at the same time (N) increases the performance gain decreases. Note that S must be less than 1 by definition of *Amdahl's Law*.

A probabilistic model would of course be more accurate to explain the behaviors observed in the experiment. However, our lack of knowledge about the computer internals (current processes running, hardware activity, memory state, etc.) prevents us from constructing such a model.

Conclusion

As a result of our experiment, we conclude that an increase in the number of threads is not enough to increase the performance without an increase in the number of mutex locks. This made us aware of the fact that synchronization suppresses the overall performance gained by using threads. In a synchronization problem, one must be very careful to tune the parameters which determine how many threads can execute and how many resources can be allocated to these threads at an instance.

References

[1] Silbertschats, Galvin, and Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018, p. 164.

[2] ——, *Operating System Concepts*, 10th ed. Wiley, 2018, p. 257.