
COL818 : Principles of Multiprocessing Systems

Name: Vatsal Jingar
Entry Number: 2020CS50449

Date: 8 April 2024
Assignment: 2

1 Build the Code

To run the code execute,

- make all

The code will run the code for tas, ttas, anderson, CLH and MCS locks.

out.txt contains all the data collected by benchmarking. The figure 1 shows the execution of code.

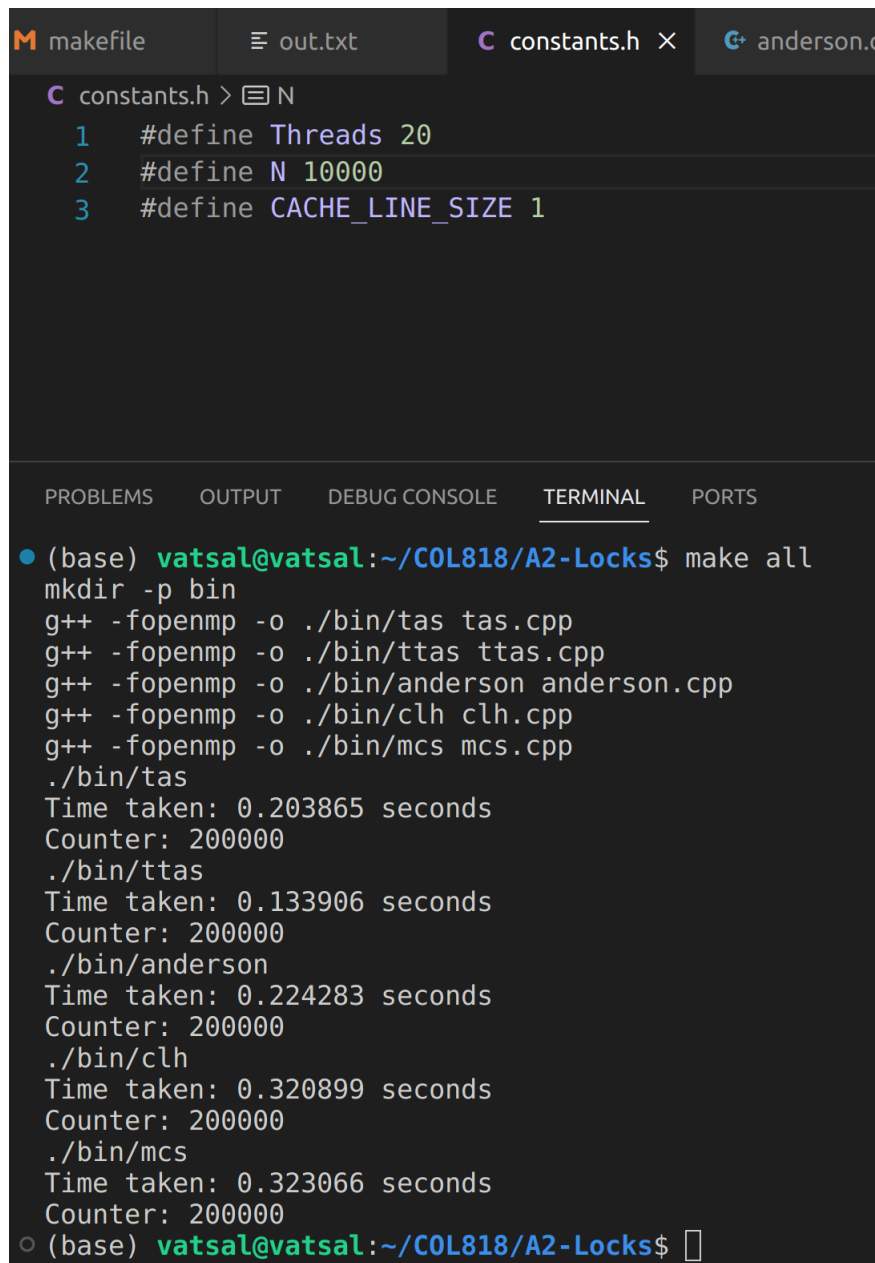
All the locks are implemented in the way, they are explained in Hilhery (The art of Multicore). For implementing atomic methods, I have atomic library of cpp.

1.1 Critical Points

- In implementation of MCS lock, while using `compare_exchange_strong`, I found that when the exchange is not successful it was not returning the right address. Hence I have used a different hack to just store the address before trying this method.
- All the atomic methods were executed by strictly following sequentially consistent instructions.
- Anderson Lock uses 1-D array as queue to decrease the contention.
- CLH lock removes the use of Global queue. Instead it distributes the queue locally. Due to use of linked list by pointers, it also avoids false sharing.
- For each lock, there are two methods as default. And are implemented in similar way shown in the book.

2 False Sharing in Anderson Lock

Due to false sharing and high number of cache misses, Anderson lock was not performing well. Although, to avoid this, I performed cache alignment. I also avoided using vectors and instead used static arrays. For threads = 8 , when I performed the latency testing, for anderson lock it shown 0.38 seconds. When false sharing is performed this value dropped down to 0.25 seconds. To perform the false sharing, we need to change the mechanism of lock by adding cacheline size in slot instead of 1. But as anderson lock do not consider this modification, we will perform benchmarking using original anderson lock.



```
makefile  out.txt  constants.h  x  anderson.c

C constants.h > N
1  #define Threads 20
2  #define N 10000
3  #define CACHE_LINE_SIZE 1

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

• (base) vatsal@vatsal:~/COL818/A2-Locks$ make all
mkdir -p bin
g++ -fopenmp -o ./bin/tas tas.cpp
g++ -fopenmp -o ./bin/ttas ttas.cpp
g++ -fopenmp -o ./bin/anderson anderson.cpp
g++ -fopenmp -o ./bin/clh clh.cpp
g++ -fopenmp -o ./bin/mcs mcs.cpp
./bin/tas
Time taken: 0.203865 seconds
Counter: 200000
./bin/ttas
Time taken: 0.133906 seconds
Counter: 200000
./bin/anderson
Time taken: 0.224283 seconds
Counter: 200000
./bin/clh
Time taken: 0.320899 seconds
Counter: 200000
./bin/mcs
Time taken: 0.323066 seconds
Counter: 200000
○ (base) vatsal@vatsal:~/COL818/A2-Locks$
```

Figure 1: Code Execution

3 Benchmarking Style

To benchmark the locks, I have a shared counter method. In this method, every thread tries to increase the counter n times results in final value as $n \times \text{numThreads}$. For every lock, only the lock type is changed and remaining code is similar so that cache misses could be identical with respect to bench-marking code and hence we can analyse the latencies of lock precisely. The method is justifiable as it avoids the random noise that can take place due to any unknown parameters. Increasing n would give a more precise latency output. The following is code for it. Increasing shared counter involves pure contention for entering in critical section and do not depend on any other parallelism behaviour. Therefore, I assume this is as a good way of analysing locks.

Lock Initialization

```
ALock lock1(Threads);  
int counter = 0;
```

Shared counter Increment

```
void increment(int n, Threadlocal* local) {  
    for (int i = 0; i < n; i++) {  
        lock1.lock(local);  
        counter++;  
        // cout << counter << endl;  
        lock1.unlock(local);  
    }  
}
```

Driver Code

```
int main(){  
    int n = N;  
    int num_threads = Threads;  
    vector<Threadlocal> locals(num_threads);  
  
    auto start = chrono::high_resolution_clock::now();  
    #pragma omp parallel num_threads(num_threads)  
    {  
        int tid = omp_get_thread_num();  
        increment(n, &locals[tid]);  
    }  
    auto end = chrono::high_resolution_clock::now();  
    chrono::duration<double> elapsed = end - start;  
  
    cout << "Time taken: " << elapsed.count() << " seconds" << endl;
```

```

    cout << "Counter: " << counter << endl;

    return 0;

}

```

4 Final Observations

The following shows the plot for latencies for described benchmarking code versus number of threads. Y axis depicts latencies in **seconds** while X axis depicts number of threads. Each thread tries to increase counter 10000 times which is a reasonable number to avoid differences that can occur due to random noise and understand the actual contention that is happening where each thread acquires a lock 10000 times.

4.1 Discussion

As I had only 18 hardware threads, I have executed only till 18 hardware threads. However, I found that 18 was a sufficient number to analyse the average latencies.

1. In general, even after having false sharing in anderson lock, it has shown better efficiency for high number of threads. This is because, after a particular number, the overhead of contention increases greater the overhead of contention to enter in critical section. And hence for higher number of threads, anderson lock latency was found lesser than tas and ttas which was theoretically proven in class.
2. For small number of threads, no pattern was observed. But as the number of threads increased, the MCS and CLH started to show high efficiency by maintaining the latency. While TAS and TTAS latencies were constantly increasing and hence shown low efficiency for high contention.
3. As visible from plot 2, MCS and CLH has high efficiency as latency is almost constant against increasing number of threads (eventual increase in contention).

```

vatsal@vatsal: ~
(base) vatsal@vatsal:~$ lscpu | grep cache
L1d cache: 544 KiB (14 instances)
L1i cache: 704 KiB (14 instances)
L2 cache: 11.5 MiB (8 instances)
L3 cache: 24 MiB (1 instance)
(base) vatsal@vatsal:~$ getconf LEVEL1_DCACHE_LINESIZE
64
(base) vatsal@vatsal:~$

```

Figure 3: Cache Line

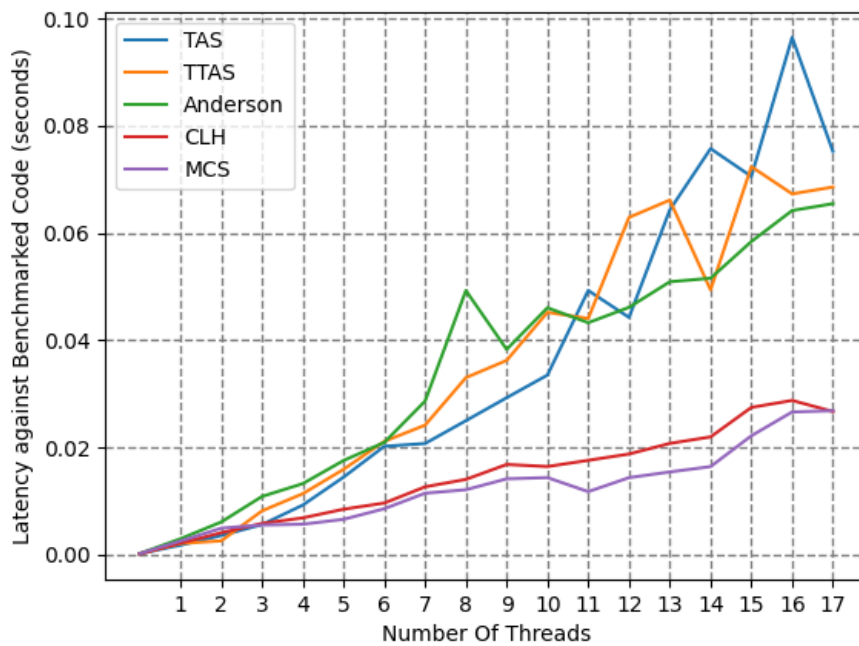


Figure 2: Latencies on Benchmarked code v/s Nummber of Threads

4.2 Increasing Performance of Anderson Lock

Instead of using an flag array of bool, I used array of integers. After checking the L1 cache line size of my device, I found it was 64 bytes. And making it int flags would decrease the contention. I have added one extra file for this purpose where the flag array uses int. The performance was significantly boosted by this change.