CS433: Parallel Programming
Assignment II Report

Aditi Goyal - 190057
Yatharth Goswami - 191178

April 3, 2022

Contents

I Locks

In this problem, we were asked to implement various locks and compare their performances on different number of threads. The acquire release functions are in `sync_library.cpp` and the main functions are in `pthread_main_lock.cpp`. Openmp's lock implementation is in `omp_main_lock.cpp`. Given below is the compilation procedure for different kind of locks.

1.1 Compilation Procedure

1.1.1 Lamport's Bakery Lock

```
g++ -O3 -pthread -DBAKERY pthread_main_lock.cpp -o pthread_main_lock
```

1.1.2 Spin lock

```
g++ -O3 -pthread -DSPIN_LOCK pthread_main_lock.cpp -o pthread_main_lock
```

1.1.3 Test-and-test-and-set

```
g++ -O3 -pthread -DTTS pthread_main_lock.cpp -o pthread_main_lock
```

1.1.4 Ticket lock

```
g++ -O3 -pthread -DTICKET_LOCK pthread_main_lock.cpp -o pthread_main_lock
```

1.1.5 Array lock

```
g++ -O3 -pthread -DARRAY_LOCK pthread_main_lock.cpp -o pthread_main_lock
```

1.1.6 POSIX Mutex

```
g++ -O3 -pthread -DPTHREAD_MUTEX pthread_main_lock.cpp -o pthread_main_lock
```

1.1.7 Binary semaphore

```
g++ -O3 -pthread -DPTHREAD_SEMAPHORE pthread_main_lock.cpp -o pthread_main_lock
```

1.1.8 Openmp

```
g++ -O3 -fopenmp omp_main_lock.cpp -o omp_main_lock
```

To run the executable for a given number of threads, the command is:
`./EXECUTABLE_NAME NUMBER_OF_THREADS`

1.2 Performance Evaluation

The code was run on the cse machine 172.27.30.118 (16 cores). Here is the overall experimental setup:

- The execution time is only taken for the solve function.
- The execution times are measured for thread count = 1, 2, 4, 8 and 16.
- The execution times were measured as average of 10 executions.

	Bakery	Spin	TTS	Ticket	Array	POSIX Mutex	Semaphore	Openmp	Best
1	285312	86266	96033	95659	148826	192279	176131	139905	Spin
2	2715314	1621645	493496	1431223	2861106	981978	1846849	1128429	TTS
4	5952073	9254157	1597532	3832086	5642428	2029668	6679376	2771792	TTS
8	16356033	31126788	5968446	10471206	11463403	5527384	16388949	14118589	POSIX Mutex
16	55654724	92362024	18891660	21535530	22024113	12283329	32997691	27687461	POSIX Mutex

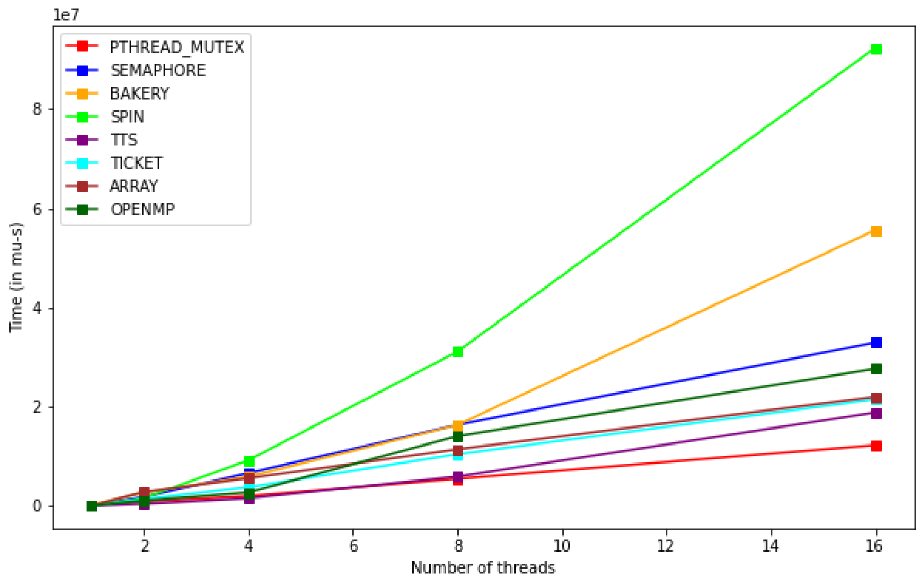

Figure 1: Time taken vs Number of threads for different locks

1.3 Observations and Explanation

- The comparison between different locking algorithms makes more sense as we increase the number of threads. Consider the case of 1 thread, there is no contention, hence no busy waiting/unsuccessful attempts at acquiring lock. So, the performance is somewhat similar.

- We can see that for large number of threads the POSIX Mutex lock performs the best while for smaller number of threads primitive hardware locks like TTS perform better than POSIX Mutex.

- POSIX Binary Semaphore performs worse as as compared to POSIX Mutex since it has other overheads like maintaining counters as well.

- We can see that in general the locks built using hardware supports are performing much better as compared to software implemented Bakery's lock.

- Test and Test and Set (TTS) performs much better than pure Spin lock as suggested in the theory as well. Reason being the fact that our implementation of Spin lock used atomic cmpxchgl instruction to check if lock is free or not and on the other hand Test and Test and Set just used simple load instructions to check if the lock is free.

- For thread count upto 16, we observe that ticket and array based locks perform a little worse than TTS lock. This maybe due to the fact that advantage of less traffic in ticket and array lock may become dominant as we increase the number of threads. The poorer performance of Ticket and Array lock might also be due to the fact that our Fetch and Inc is built using cmpxchgl instruction and this implementation requires a while loop until the exchange becomes successful. Even after all this the effect of traffic and fairness comes into picture as we increase the number of threads. We can see that on increasing the number of threads the difference between execution times of test and test and set and ticket lock starts decreasing, which should be the case theoretically as well and ticket lock may outperform test and test and set.

- Our implementation of array lock performs slightly poorly as compared to the Ticket based lock. This maybe because of the fact that for smaller number of threads the other implementation overheads for array lock dominates. We can although see that for large number of threads (8, 16); the difference between the execution times start decreasing. This maybe due to much less traffic in array based locks than ticket locks.

II Barriers

In this problem, we were asked to implement various barriers and compare their performances on different number of threads. The barrier functions are in `sync_library.cpp` and the main functions are in `pthread_main_barrier.cpp`. Openmp's barrier implementation is in `omp_main_barrier.cpp`. Given below is the compilation procedure for different kind of barriers.

2.1 Compilation Procedure

2.1.1 Centralized sense reversal barrier (busy wait)

```
g++ -O3 -pthread -DSENSE_REVERSAL pthread_main_barrier.cpp -o
pthread_main_barrier
```

2.1.2 Tree barrier (busy wait)

```
g++ -O3 -pthread -DTREE_BUSY_WAIT pthread_main_barrier.cpp -o
pthread_main_barrier
```

2.1.3 Centralized barrier (POSIX condition variable)

```
g++ -O3 -pthread -DCENTRALISED_CONDITIONAL pthread_main_barrier.cpp -o
pthread_main_barrier
```

2.1.4 Tree barrier (POSIX condition variable)

```
g++ -O3 -pthread -DTREE_CONDITIONAL pthread_main_barrier.cpp -o
pthread_main_barrier
```

2.1.5 POSIX barrier interface

```
g++ -O3 -pthread -DPTHREAD_BARRIER pthread_main_barrier.cpp -o
pthread_main_barrier
```

2.1.6 Openmp barrier

```
g++ -O3 -fopenmp omp_main_barrier.cpp -o omp_main_barrier
```

To run the executable for a given number of threads, the command is:
`./EXECUTABLE_NAME NUMBER_OF_THREADS`

2.2 Performance Evaluation

The code was run on the cse machine 172.27.30.118 (16 cores). Here is the overall experimental setup:

- The execution time is only taken for the solve function.
- The execution times are measured for thread count = 1, 2, 4, 8 and 16.
- The execution times were measured as average of 10 executions.

	Sense reversal	Tree (busy)	Centralized (condition)	Tree (condition)	POSIX barrier	Openmp	Best
1	42264	6211	25562	8787	410318	412794	Tree (busy)
2	554464	146842	5222338	11051608	3556636	439797	Tree (busy)
4	1397414	418448	11863156	22371726	10363213	470485	Tree (busy)
8	8244213	694526	28195655	36718106	18627291	621878	Openmp
16	27966944	998698	65636505	52403483	36762336	741349	Openmp

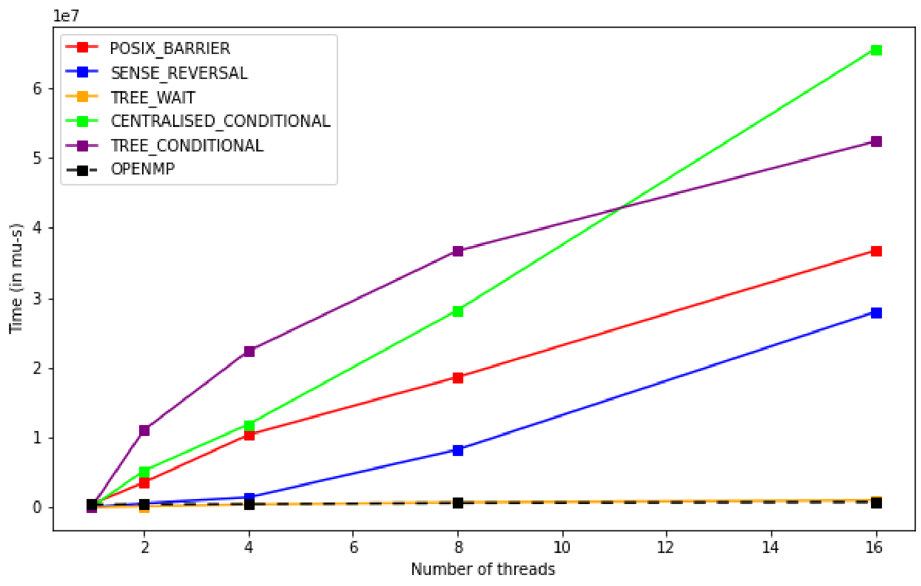

Figure 2: Time taken vs Number of threads for different barriers

Note: Openmp results are shown with dashed line to differentiate tree busy wait barrier and openmp because at this scale, they look quite similar.

2.3 Observations and Explanation

- The comparison between different locking algorithms makes more sense as we increase the number of threads.
- For large number of threads Openmp's barrier implementation performs the best while for smaller number of threads tree based barrier work the best.
- Tree busy waiting barrier algorithm performs better than sense reversal barrier algorithm as theoretically expected. It is due to the fact that the tree barrier implementation involved no locks as such and also the fact that it shows much better scalability as we increase the number of threads. This is due to the inherently parallel nature of the algorithm.
- Implementations using conditional variables of both the centralized and tree algorithm take more time as compared to their busy wait implementations. This maybe due to the fact that the conditional variable implementation use a lot of locks which makes them heavy.
- We can see that since the tree barrier algorithm is inherently parallelisable, the ratio of execution times of consecutive thread count for tree barrier's conditional barrier decreases on increasing thread number. At large thread

counts (16), tree conditional becomes better than centralised conditional, even though it uses more locks than centralized conditional.

- The openmp barrier implementation scales really well with increasing number of threads. We can see that their execution times are almost constant for all thread counts.