CS610: Programming for Performance
Assignment 2

Wadkar Srujan Nitin
221212

Solution 1

Setup

Input is blocked by taking an element from the block and updating its contribution to all the output cells to which it can contribute, by multiplying the appropriate cell of the kernel with it. To run the code, you first need to compile and then execute:

```
make
./bin/problem1.out blockHeight blockWidth blockDepth
```

The code was run on `csews9` for input size $(1 \ll 9) \times (1 \ll 9) \times (1 \ll 9)$ with different block parameters, and the results are shown below.

BH	BW	BD	Naive			Blocked Input		
			L1 DCM	L2 DCM	Time (us)	L1 DCM	L2 DCM	Time (us)
1	1	1	3.32e8	1.50e8	1.23e6	1.65e8	1.51e8	3.96e6
1	1	2	3.37e8	1.51e8	1.24e6	1.63e8	1.54e8	3.44e6
1	1	4	3.40e8	1.55e8	1.23e6	1.59e8	1.52e8	3.21e6
1	1	32	3.38e8	1.53e8	1.24e6	1.52e8	1.50e8	3.01e6
1	1	64	3.42e8	1.51e8	1.23e6	1.52e8	1.51e8	2.98e6
1	1	128	3.36e8	1.51e8	1.25e6	1.52e8	1.52e8	2.99e6
1	1	512	3.41e8	1.53e8	1.22e6	1.51e8	1.50e8	2.97e6
1	2	2	3.37e8	1.52e8	1.22e6	2.41e8	1.48e8	3.29e6
2	2	2	3.36e8	1.52e8	1.23e6	2.41e8	1.48e8	3.27e6
4	4	4	3.37e8	1.53e8	1.23e6	2.04e8	1.49e8	3.02e6
8	8	8	3.40e8	1.52e8	1.23e6	1.61e8	1.52e8	2.97e6
16	16	16	3.38e8	1.53e8	1.23e6	1.52e8	1.50e8	2.93e6
32	32	32	3.36e8	1.52e8	1.24e6	1.52e8	1.50e8	2.91e6

Table 1: Performance metrics (BH = Block Height, BW = Block Width, BD = Block Depth).

The data shows that we get a saturation in the improvement of L1 cache misses after block size $1 \times 1 \times 32$, and the different block parameters do not provide much improvement compared to $1 \times 1 \times 32$. However, the L2 cache misses remain almost the same, suggesting that the block already fits in L1.

The time, however, is worsened. The possible reasons for this could be the fact that the naive version in effect behaves like the output has been blocked, as it computes the entire value of a cell at once and then writes the data to it. Also, when adding the contribution to different output values, consider the case when we need to evict some values to accommodate the next block. Thus, if the values in the cache lines are not computed and the cache is write-back, we are sending unnecessary write-back messages to the higher-level cache.

Whereas in the naive version, this is not the case: we bring the cache line with only read permission, and even if there are evictions, we do not need to do any write-backs and thus reducing on the time taken by the program.

Solution 2

Setup

The program was run with 10 threads on `csews9`, each working on a separate file. Each file contains 10^6 lines, and each line has 4 words. To run the code, you first need to compile and then execute:

```
make
perf c2c record -F 30000 -u -- ./bin/problem2.out 10 ./prob2-test2/input
perf c2c report -NN --stdio -i perf.data
```

Initial Profiling

We first examine the `perf` output of the given program to identify performance bottlenecks.

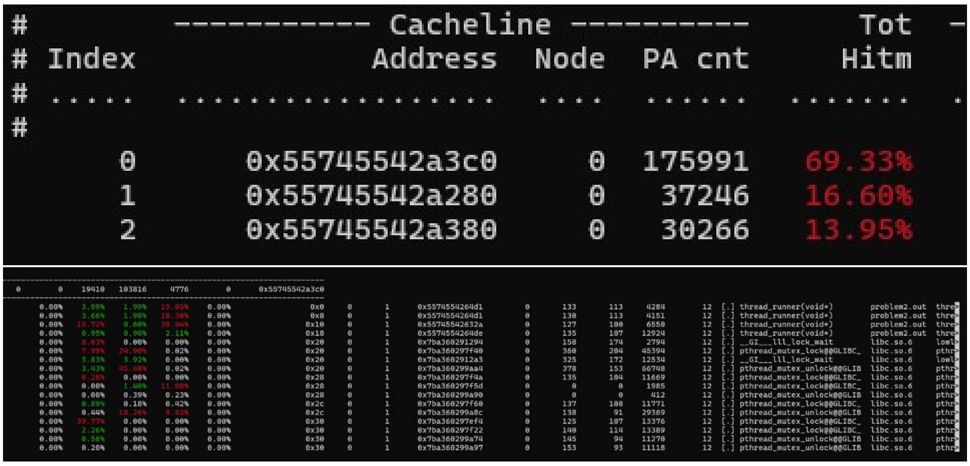

Figure 1: Initial perf output showing contention.

From the first report, we observe that contention for cache line 0 is significant. A more detailed report about the cache line shows that the major bottleneck is due to the use of locks.

First Optimization: Reducing True Sharing

The first optimization targets decreasing the use of locks (i.e., reducing true sharing overhead). Instead of updating the shared structure directly, each thread maintains a local count of processed words and processed lines. These local counts are added to the shared structure only once, after the thread finishes processing its file.

Profiling After First Optimization

When we run `perf` again, some cache line contention still remains:

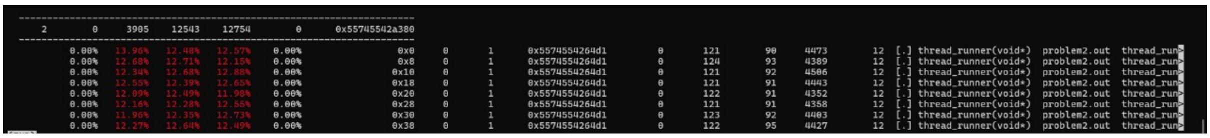

Figure 2: perf output after optimisation

The contention is now primarily due to the `word_count` array being accessed by multiple threads, leading to false sharing.

Second Optimization: Reducing False Sharing

To eliminate false sharing, we use the local word counts that were already computed by each thread. These are written into the shared structure only after the thread completes processing its file. This ensures that different threads do not update adjacent elements of the shared array simultaneously.

Final Profiling

After applying this optimization, we run `perf` once more. No cache line contention is observed, indicating that both true and false sharing issues have been eliminated.

Performance Results

- Unoptimized version: ~ 5 seconds
- After removing true sharing: ~ 0.65 seconds
- After removing false sharing: ~ 0.2 seconds

Solution 3

Setup

To run the code, you first need to compile and then execute:

```
make
./bin/problem3.out
```

The number of threads can be changed by changing the `NUM_THREADS` variable and the `MAX_NUM_THREADS` variable

Lock Type	**1**	**2**	**4**	**8**	**16**	**32**
Pthread mutex	25	355	1469	1959	10103	64796
Filter lock	8	612	4348	110702	978110	17693672
Bakery lock	24	492	6944	47129	318102	1703055
Spin lock	24	48	2816	15866	126066	1148529
Ticket lock	14	22	6438	32218	165726	530633
Array Q lock	20	177	2880	35259	353130	703587

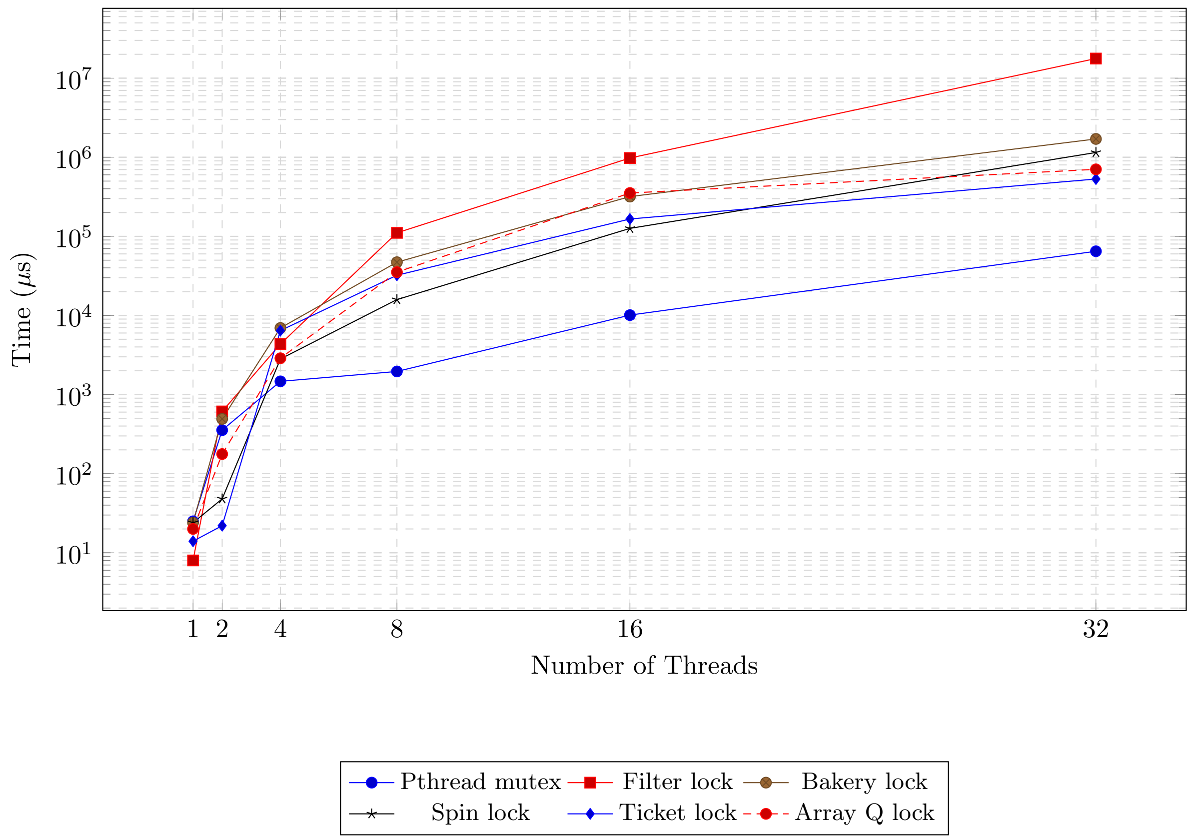

Figure 1: Execution time of different locks vs number of threads (in microseconds)

Observations and Results

The code was executed on `image1.cse.iitk.ac.in`, which is a machine with around 40 cores. The column corresponding to 64 threads was skipped, as the process did not finish within an acceptable time limit.

A similar issue was observed when running the code with 16 threads on the `csews` systems, which have only 12 cores. This suggests that the exceptionally large execution time could be caused by insufficient core availability rather than inefficiency in the implementation itself.

Filter Lock Behavior

From the results, we observe that the **Filter lock** consistently takes significantly larger time compared to other locks. A possible explanation comes from the algorithm itself: in the Filter lock, a thread must progress through `MAX_NUM_THREADS` levels, and at each level it must busy-wait by checking the state of all other threads before proceeding to the next level. This introduces substantial overhead compared to other locks, particularly those based on hardware primitives that do not require checking the state of every other thread. Hence, the Filter lock exhibits the worst performance.

Other Custom Locks vs Pthread Mutex

The other custom locks (Bakery, Spin, Ticket, and Array Q lock) show somewhat comparable performance in terms of runtime. However, as the number of threads increases, the **pthread_mutex** implementation consistently outperforms all custom locks.