CS610 Assignment-2

Mahaarajan J -220600

September 2, 2025

1 Problem-1

- Implemented a blocked version of the function `cc_3d_no_padding`
- Experimented with various block sizes by trying all combinations of $(block_x, block_y, block_z)$ where each $block_i$ is tried from $\{1, 2, 4, 6, 8, 16, 32\}$
- The PAPI counter used to analyze cache misses was `PAPI_L2_TCM`
- We run each configuration 5 times take the average and report the configuration with the best time.
- The dimensions of input vector is made 128*128 while the kernel was kept the same.
- **Naive Version**
  - Average time taken (5 runs): 0.143970s
  - Cache misses: 2212767
- **Blocked Version (best average time)**
  - Config: (32,8,32)
  - Average time taken (5 runs): 0.165944
  - Cache misses: 3026446
- **Blocked Version (least misses)**
  - Config: (4,2,4)
  - Average time taken (5 runs): 0.174191
  - Cache misses: 1339860
- The results obtained varied across multiple runs with such different optima for average time and misses even though theoritically it needs to be same, the reason could be noise and increased code size due to adding of extra loops in the blocked version
- Additionally the Naive version performs better. This could again be due to smaller array sizes as compared to the L2 cache
- **Running instructions**
  - `g++ 220600-prob1.cpp -lpapi -o prob1` (Compilation)
  - `runner_prob1.sh` (runs and checks all possibilities)
  - `python3 anal_prob1.py` (gives out the best result)

2 Problem-2

Parameters

- Used a total of 15 input files with each set of 5 files a 500 times replica of the given 5 input files
- Used 5 threads in parallel

Benchmarking the unoptimised implementation

- executed the binary file using perf `perf c2c record -F 60000 prob2_naive 5 prob2-test1/input`
- Observed that the time taken was **0.016363s** and on using `perf c2c report --stdio` observed:
- Cache line `0xffff923d418b50c0` had a **22.78%** HITM which meant that there was false sharing in the application

Identifying and Correcting bugs

- The performance bottleneck was due to multiple threads updating adjacent elements in a shared data structure `tracker.word_count[]` without any padding, causing frequent invalidations of the cache line and leading to false sharing.
- To eliminate false sharing, each thread's counter was moved into a structure padded to the size of a cache line:

```
struct alignas(64) PaddedCounter {
    uint64_t val;
    char padding[64 - sizeof(uint64_t)];
};
```

- The array `word_count[NUM_THREADS]` was then declared as `PaddedCounter word_count[NUM_THREADS];`, ensuring that each thread writes to a separate cache line.
- This change removed the high HITM percentage in subsequent runs (observed ¡ 0.5% HITM), confirming the removal of false sharing.
- Additionally, updates to total word count were done using a thread-local accumulator and then combined with a mutex-protected critical section to reduce contention on shared variables.

```
while ((pos = line.find(delimiter)) != std::string::npos) {
  token = line.substr(0, pos);
  tracker.word_count[thread_id].val++;
  local_word_count++;
  line.erase(0, pos + delimiter.length());
}
//Add total word_count together
pthread_mutex_lock(&tracker.word_count_mutex);
tracker.total_words_processed += local_word_count;
pthread_mutex_unlock(&tracker.word_count_mutex);
```

Benchmarking the optimised implementation

- executed the binary file using perf `perf c2c record -F 60000 prob2_opt 5 prob2-test1/input`
- Observed that the time taken was **0.007206s** and on using `perf c2c report --stdio` observed all user_space cache-lines had $< 10\%$ Hence we have avoided false sharing and get about 30% speedup

Running Instructions

- `g++ 220600-prob2.cpp -o prob2_opt` (Compilation)
- `./prob2_opt 5 prob2-test1/input`

3 Problem-3

Implementation details

- **Inline Assembly for Atomic Operations:**
  - `xchg`: Used to implement spin-based mutual exclusion (`atomic_lock_test_and_set`).
  - `lock xadd`: Used for atomic increment/fetch-add operations (`atomic_fetch_and_add`).
  - `lock cmpxchg`: Used for compare-and-swap operations (`atomic_compare_and_swap`).
  - These operations allow direct control over synchronization without relying on higher-level abstractions like `std::atomic`

- **Memory Fencing:**
  - A full memory fence was enforced using the `mfence` instruction via `asm volatile ("mfence" ::: "memory")` in a function `memory_barrier()`.
  - This ensures that memory operations are not reordered by the compiler or CPU across critical sections.
- **Spinning Optimization:**
  - The `pause` instruction was used in spin loops as a part of `thr_wait()` function defined
- **False Sharing Avoidance:**
  - Custom structs such as `PaddedInt`, `PaddedBool`, and `FlagSlot` were defined using `alignas(64)`.
  - This ensures that independent per-thread variables do not reside on the same cache line, thus eliminating false sharing.

Benchmarking Setup

- Each lock was benchmarked by measuring the total time taken by all threads to complete their `N = 1e7` iterations with a timeout of 2 minutes (Number of threads 1,2,4,8,16,32)
- Timing data was recorded in microseconds for fair comparison across different lock types.

Comparision of various locks

Table 1: Time Taken (in μs) vs Number of Threads

Lock Type	**1**	**2**	**4**	**8**	**16**	**32**
Pthread	162	2428	9273	69579	305089	1610647
Filter	241	5822	30511	238082	Timeout	Timeout
Bakery	748	5618	22476	116410	Timeout	Timeout
Spin	472	2159	22573	137916	622618	7347838
Ticket	519	6356	26665	102392	Timeout	Timeout
ArrayQ	510	6997	29712	115144	Timeout	Timeout

Analysis:

- As the number of threads increases, all locks show a significant rise in execution time due to increased contention. This is expected in highly concurrent environments.
- **Pthread mutex**, being implemented at the OS level, shows stable and consistent performance across all thread counts, successfully completing even at 32 threads.
- **Filter**, **Bakery**, **Ticket**, and **ArrayQ** locks perform reasonably well up to 8 threads but fail to scale beyond 16 threads, leading to timeouts. These locks involve per-thread coordination structures which cause severe performance degradation due to high contention.
- **Spin lock** works well for low thread counts but exhibits extremely poor scalability. Performs better than the other locks and is beaten only by PThread
- Overall, the **Pthread mutex** is the most robust under high contention, while **Spin** and user-level ticket-based locks work efficiently only in low-thread scenarios.

Running Instructions

- `g++ 220600-prob3.cpp -o prob3` (Compilation set number of threads accordingly)
- `./prob3`
- `./script.sh` (to see individual locks with timeout)