The program efficiently processes 100 items using N producer-consumer pairs, ensuring synchronization and parallel execution. The workload is evenly distributed among threads, and condition variables prevent busy-waiting, making execution efficient.

Producers: Generating and Inserting Data into the Buffer

- Each producer generates a random number using `generateRandomNumber()`.
- It checks if TOTAL_ITEMS have already been produced. If so, it stops.
- It waits if the buffer is full using `pthread_cond_wait(&shared->buffer_not_full, &shared->buffer_mutex)`.
- Once space is available, it adds the number to the circular buffer (protected by `buffer_mutex`).
- After inserting data, it signals consumers using `pthread_cond_signal(&shared->buffer_not_empty)`, allowing them to proceed.
- A small delay (`usleep(50)`) allows natural OS-level context switching.

Consumers: Retrieving and Processing Data

- Each consumer waits if the buffer is empty using `pthread_cond_wait(&shared->buffer_not_empty, &shared->buffer_mutex)`.
- It retrieves a number from the buffer and updates `out` index.
- After consuming, it signals producers using `pthread_cond_signal(&shared->buffer_not_full)`, allowing them to add more items.
- It computes the cumulative sum and average for its assigned workload.

- PRF RESULT FOR 2 PRODUCER-CONSUMER THREADS:

```
Performance Results
Total execution time: 6.46 ms
Total items processed: 100
Throughput: 15473.06 items per second
# started on Sun Mar  2 19:45:13 2025


 Performance counter stats for './p 2':

         5,112,142      cycles
         4,319,040      instructions                     #    0.84  insn per cycle
           809,452      branches
           339,434      cache-references
            74,389      cache-misses                     #   21.92% of all cache refs
               218      context-switches

       0.008404792 seconds time elapsed

       0.000000000 seconds user
       0.006452000 seconds sys


# started on Sun Mar  2 19:45:13 2025


 Performance counter stats for './p 2':

         1,016,376      L1-dcache-loads
            80,583      L1-dcache-load-misses            #    7.93% of all L1-dcache accesses
            23,278      LLC-loads
            16,931      LLC-stores

       0.008597713 seconds time elapsed

       0.001139000 seconds user
       0.005699000 seconds sys
```

- PERF RESULT FOR 4 PRODUCER-CONSUMER THREADS:

```
Performance Results
Total execution time: 3.90 ms
Total items processed: 100
Throughput: 25668.62 items per second
# started on Sun Mar  2 19:45:13 2025


 Performance counter stats for './p 4':

         7,122,664      cycles
         5,273,579      instructions                     #    0.74  insn per cycle
           987,422      branches
           557,604      cache-references
            69,573      cache-misses                     #   12.48% of all cache refs
               244      context-switches

       0.006112044 seconds time elapsed

       0.001583000 seconds user
       0.007917000 seconds sys


# started on Sun Mar  2 19:45:13 2025


 Performance counter stats for './p 4':

         1,207,603      L1-dcache-loads
           108,537      L1-dcache-load-misses            #    8.99% of all L1-dcache accesses
            34,035      LLC-loads
            23,886      LLC-stores

       0.005663627 seconds time elapsed

       0.002536000 seconds user
       0.005072000 seconds sys
```

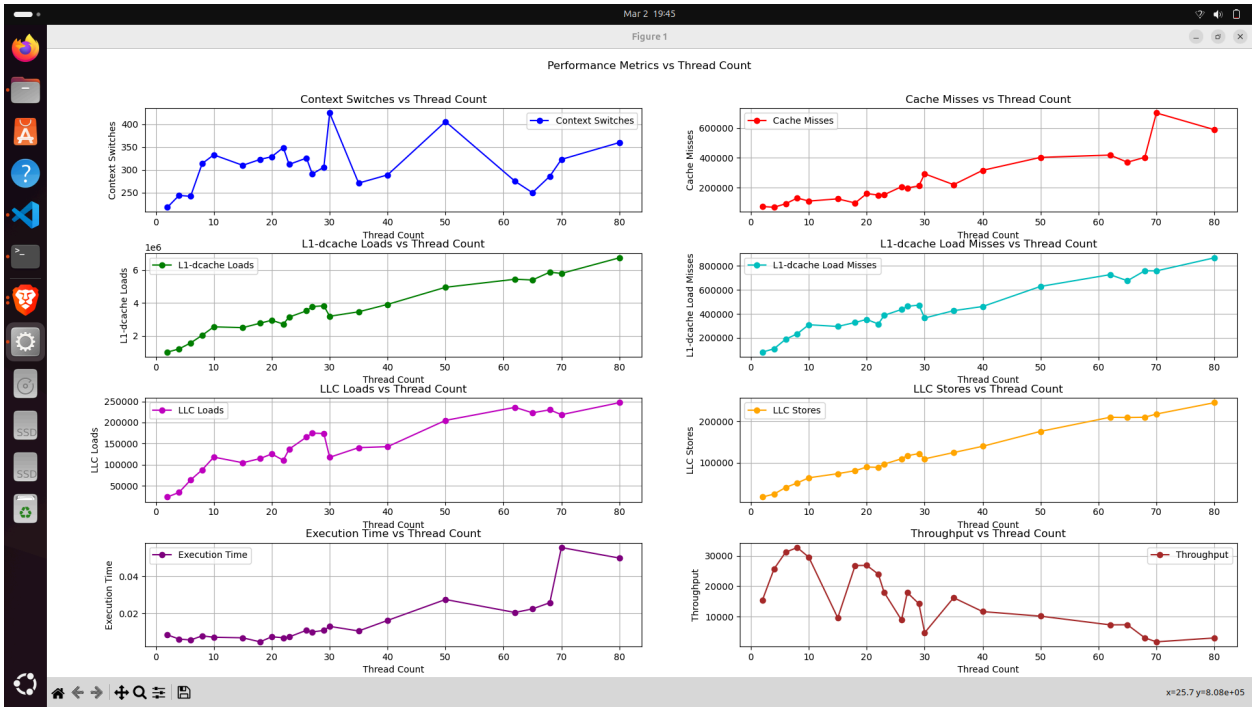
- PERF RESULT FOR 50 PRODUCER-CONSUMER THREADS:

```
Performance Results
Total execution time: 9.85 ms
Total items processed: 100
Throughput: 10150.96 items per second
# started on Sun Mar  2 19:45:15 2025


 Performance counter stats for './p 50':

     34,392,825      cycles
     21,655,379      instructions                     #    0.63  insn per cycle
      3,997,230      branches
      2,541,593      cache-references
        403,369      cache-misses                     #   15.87% of all cache refs
            406      context-switches

    0.027536233 seconds time elapsed

    0.004912000 seconds user
    0.024563000 seconds sys


# started on Sun Mar  2 19:45:15 2025


 Performance counter stats for './p 50':

     4,950,649      L1-dcache-loads
       629,376      L1-dcache-load-misses            #   12.71% of all L1-dcache accesses
       205,054      LLC-loads
       175,661      LLC-stores

    0.017503353 seconds time elapsed

    0.004244000 seconds user
    0.024618000 seconds sys
```

- PERF RESULT FOR 80PRODUCER-CONSUMER THREADS:

```
Performance Results
Total execution time: 33.42 ms
Total items processed: 100
Throughput: 2992.50 items per second
# started on Sun Mar  2 19:45:17 2025


 Performance counter stats for './p 80':

     49,435,617      cycles
     29,663,211      instructions                     #    0.60  insn per cycle
      5,436,859      branches
      3,612,909      cache-references
        588,887      cache-misses                     #   16.30% of all cache refs
            360      context-switches

    0.049972131 seconds time elapsed

    0.006769000 seconds user
    0.033002000 seconds sys


# started on Sun Mar  2 19:45:17 2025


 Performance counter stats for './p 80':

      6,738,869      L1-dcache-loads
        868,296      L1-dcache-load-misses            #   12.88% of all L1-dcache accesses
        247,046      LLC-loads
        245,089      LLC-stores

    0.026929515 seconds time elapsed

    0.003453000 seconds user
    0.029355000 seconds sys
```

Experimenting by varying the numbers of producer and consumer threads:



Performance Analysis Based on Graphs

1.The **context switches** increase as more threads are introduced, which indicates that the CPU is actively switching between tasks. Initially, context switches rise due to increased CPU utilization. However, as the number of threads continues to grow, the OS attempts to optimize scheduling, leading to fluctuations. There are noticeable spikes around 30 and 50 threads, likely caused by thread contention. The increasing context switches suggest that the CPU is being utilized more efficiently compared to previous implementations, where context switching remained low despite a high thread count.

2.The **cache misses** increase significantly with more threads, showing that multiple threads are competing for cache space, leading to frequent evictions. As the CPU fails to retrieve data from the cache, it is forced to access slower main memory, increasing execution time. Similarly, L1-dcache loads and load misses also rise, confirming that the cache is overwhelmed with requests. As the L1 cache becomes inefficient, data retrieval falls back to slower memory hierarchies, impacting overall system performance.

3.The **Last Level Cache (LLC) loads and stores** continue to rise with increasing thread count. This is an indication that L1 and L2 cache misses are frequent, leading to more accesses at the LLC level. While LLC helps mitigate cache misses, it is still significantly slower than L1 and L2 caches, adding to execution latency. More threads accessing LLC indicate that memory contention is growing, further reducing CPU efficiency.

4.**Execution time**, which should ideally decrease with more threads, instead increases. The reason for this is the combined effect of high context switching, cache contention, and synchronization overhead. More threads mean more resource-sharing, leading to performance degradation due to inefficient memory access patterns. Instead of parallelizing tasks efficiently, excessive threading results in CPUs spending more time handling context switching and cache stalls rather than performing useful computations.

5. **Throughput** decreases after reaching an optimal number of threads. While adding threads initially boosts throughput, beyond a certain point, thread management overhead takes over, reducing the system's ability to process tasks efficiently. Instead of executing tasks in parallel, threads

start interfering with each other due to increased competition for CPU resources, leading to lower throughput.

Why Execution Time is Increasing Instead of Decreasing:

The main reason execution time increases rather than decreases is oversubscription. With only four CPU cores, adding more than 4-8 threads causes excessive contention, where the OS must frequently swap threads in and out. This frequent swapping, also known as context switching, wastes valuable CPU cycles that could otherwise be used for actual computation. While moderate context switching is necessary for multitasking, excessive switching results in performance degradation.

Another contributing factor is high context switching overhead. Since each thread must save and restore registers when switching, latency is introduced. More threads mean more switching, and instead of improving performance, it reduces the actual time spent processing data. Additionally, a memory bottleneck due to cache contention occurs when too many threads attempt to access shared cache memory. This leads to an increased cache miss rate, forcing the CPU to access slower main memory more frequently. More cache misses lead to more memory stalls, reducing the overall efficiency of execution.

Why Throughput is Decreasing Instead of Increasing:

The CPU contention problem arises as the number of threads increases beyond an optimal range. A four-core CPU can efficiently handle four to eight threads, but beyond this, additional threads start to compete for CPU time, resulting in performance degradation. Instead of improving throughput, more threads begin interfering with each other, reducing the overall execution efficiency.

Another major issue is lock contention. Since producer and consumer threads operate on shared buffers, they must synchronize access using mutex locks. While this prevents race conditions, it also introduces wait times for threads trying to acquire locks, causing some threads to remain idle. This reduces throughput since threads are blocked instead of actively performing work.

Lastly, cache misses and memory latency increase as the number of threads grows. With higher thread counts, the CPU frequently misses cached data, forcing it to retrieve information from RAM, which is significantly slower. More memory stalls result in fewer operations being completed per second, reducing throughput instead of increasing it.

Conclusion:

The results confirm that simply increasing the number of threads does not always improve performance. The optimal thread count should be approximately twice the number of CPU cores to achieve maximum efficiency. Beyond this limit, excessive threading introduces inefficiencies such as excessive context switching, CPU contention, and memory bottlenecks, leading to performance degradation. Cache contention also becomes a major issue, as more threads compete for limited cache space, increasing cache misses and slowing down execution.

Execution time increases rather than decreases because of frequent context switching and memory stalls, which prevent efficient CPU utilization. Similarly, throughput declines beyond an optimal thread count, as CPU cycles are increasingly wasted on thread management rather than actual processing. These results highlight the importance of carefully

balancing the number of threads with system resources to achieve optimal performance.

Optimizations :

To improve performance, the following optimizations can be applied:

1. Maintain an optimal thread count close to CPU cores × 2 (for a 4-core system, 4-8 threads work best).
2. Minimize context switching overhead by reducing excessive thread creation and avoiding unnecessary synchronization.
3. Optimize cache usage by improving memory access patterns and reducing contention for shared cache resources.
4. Use finer-grained locks instead of coarse-grained locking to reduce waiting times and improve parallelism.
5. Implement thread affinity to bind specific threads to CPU cores, reducing unnecessary thread migration.