Traffic Control Simulation: Producer-Consumer Design

This document outlines the design, data structures, and concurrency strategy for implementing the multi-threaded bounded-buffer producer-consumer pattern to simulate traffic control analysis.

1. Data Structures and Thread Safety

Data Structure	Purpose	Location	Thread-Safe?	Concurrency Strategy
TrafficData Struct	Holds one traffic signal measurement (timestamp, ID, count).	Shared & Local	Yes (Immutable)	Passed by value/const reference.
Bounded Buffer (std::queue<TrafficData>)	The shared queue between producers and consumers.	Shared	**Yes** (Blocking)	Protected by std::mutex and controlled by two std::condition_variables.
std::mutex (Buffer Lock)	Protects the bounded buffer's critical sections (push and pop).	Shared		Used with std::unique_lock for both blocking and non-blocking access.
std::condition_variable (Full/Empty)	Blocks producers when the buffer is full and consumers when the buffer is empty.	Shared		Ensures efficient waiting without busy-looping.
Congestion Map (std::map<int, long long>)	Tracks the total car count for every traffic_light_id.	Shared	**Yes** (Non-Blocking)	Protected by a separate **std::mutex (Map Lock)** for quick, non-blocking updates.
std::mutex (Map Lock)	Protects the shared Congestion Map	Shared		Ensures atomic update of traffic counts.

	during consumers' update operations.			

2. Producer-Consumer Mechanism

Bounded Buffer Implementation

The buffer is implemented using a std::queue with a fixed CAPACITY (set to 50 in the code).

- **Blocking Producer:**
    1. Acquires lock on the buffer.
    2. Waits on the **cond_producer** condition variable if the buffer is **full**.
    3. Pushes the data onto the queue.
    4. Notifies the **cond_consumer** condition variable to wake up waiting consumers.
- **Blocking Consumer:**
    1. Acquires lock on the buffer.
    2. Waits on the **cond_consumer** condition variable if the buffer is **empty**.
    3. Pops the data from the queue.
    4. Notifies the **cond_producer** condition variable to wake up waiting producers.

This design ensures that threads only consume cycles when there is work to do, preventing race conditions and deadlocks.

3. Simulation Parameters

- **Traffic Signals (X):** 20
- **Measurements per hour:** 12 (one every 5 minutes)
- **Data File:** Reads data from traffic_data.txt.
- **Top N:** The consumers collaboratively track the top 5 most congested lights.

4. Sequential Baseline vs Pthread Comparison

The measured results clearly demonstrate that the **overhead of thread management and synchronization exceeds the benefit of parallel processing** for the small dataset used.

The Cost of Concurrency (Overhead)

The Multi-Threaded model introduces significant costs that are not present in the Sequential Baseline:

1. **Thread Creation/Destruction:** Starting up and tearing down 5 threads (1 Producer+4 Consumers) takes time.
2. **Context Switching:** The operating system must constantly switch the 's focus between the Producer and Consumer threads.
3. **Synchronization Lock Contention:** Every time data is moved to or from the shared

buffer, a **mutex lock** (`g_buffer_mutex`) must be acquired and released. Similarly, the Consumer threads spend time waiting on the **condition variable** (`g_cond_consumer`).

The Missing Benefit (I/O Overlap)

The Producer-Consumer pattern is designed to overcome I/O **latency** (the time spent waiting for a hard drive to deliver data).

- Because the data file is small, the entire file is likely read into CPU cache almost instantly.
- The key benefit of I/O overlap (processing old data while waiting for new disk I/O) is zero because the slow I/O is not a factor.

Conclusion: The negligible I/O time means the multi-threaded system only experiences the **overhead** without realizing the intended **performance gain**, resulting in a net slowdown.

= Most efficient choice for small data volumes where the cost of setting up concurrent resources (threads, mutexes, condition variables) outweighs the minimal processing time.

To demonstrate true speedup, the simulation must be run with a **large data file** (e.g., millions of records) where the latency becomes the dominant factor

Blocking and Non-Blocking Operations

In MPI, the terms blocking and non-blocking refer specifically to the **communication calls** and when control returns to the program after the call is initiated.

Blocking Operations (Used Extensively)

A blocking MPI call is one that does not return until the communication is locally complete. This means:

1. **For Sending (Master):** The call returns only after the send buffer is safe to reuse.
2. **For Receiving (Slave):** The call returns only after the receive buffer contains the new data.

Your current implementation relies entirely on blocking collective operations, which ensures synchronization between processes:

MPI Function	Blocking Role in Task
`MPI_Bcast`	The Master blocks until the message (e.g., `total_hours`) is sent. Slaves block until the message is received.
`MPI_Scatter`	Used to send the exact chunk size. The Master blocks until it sends the size, and Slaves block until they receive it.

`MPI_Scatterv`	The Master blocks until it has distributed the data for the current hour. Slaves block until their assigned chunk of data is in their local buffer (`slave_local_data`).
`MPI_Gatherv`	All processes (Master and Slaves) block until the Master has successfully collected all the partial results from every other process.

Export to Sheets

Advantage in this Task: Blocking calls simplify the synchronization logic. Since the program must perform sequential hourly reports, waiting for all data to arrive before starting computation (and waiting for all results to return before starting the next hour) is the safest approach.

Non-Blocking Operations (Conceptual)

Non-blocking calls (`MPI_Isend`, `MPI_Irecv`) return *immediately*, giving the program control back while the communication proceeds in the background.

- **Absence:** Your code does not use explicit non-blocking MPI functions.
- **Non-Blocking Logic:** The most crucial **non-blocking aspect is the computation itself**. Once the Slave process finishes the blocking `MPI_Scatterv` call, it immediately starts the intense local aggregation