CS820
Minna Tran

The Magic of Sync: A Traffic Light Simulation Using Kuramoto Oscillators and Surtrac

Introduction: An Observation at the Crosswalk

It started with a simple observation while waiting to cross the street. Cars don't flow continuously like water from a tap — they arrive in distinct waves. A cluster of five or six cars would pass, then nothing for a while, then another cluster. This pattern repeated predictably.

Why waves? The answer became obvious: somewhere upstream, a traffic light had gathered these cars together at a red light, then released them as a group when it turned green. The traffic light was creating these chunks.

But this raised a deeper question. If one light creates a wave, what happens when that wave reaches the next light? If the lights aren't coordinated, the wave stops, waits, and fragments. By the time cars reach me at the crosswalk, the nice clean waves would dissolve into scattered, unpredictable trickles.

The fact that I observe clean waves with predictable gaps suggests something remarkable: the upstream lights must be *synchronized*. They're coordinating their green phases so that a wave released from one light arrives at the next light just as it turns green. This is called a "green wave," and it's no accident — it's the result of careful timing coordination between independent traffic signals.

This realization connected to something I was studying in our class – concurrency. Each traffic light operates independently, making its own decisions, running its own timing cycle — just like threads in a concurrent program. Yet somehow, they need to coordinate to achieve a global goal (smooth traffic flow) without a central controller telling each one exactly what to do.

This project explores that connection by building a traffic simulation that combines two powerful ideas: the Kuramoto model from physics (which explains how independent oscillators synchronize) and Surtrac from traffic engineering (which enables intersections to communicate and adapt to traffic demand).

Part 1: Understanding Synchronization

What is Synchronization?

Synchronization is the phenomenon where independent entities spontaneously align their behavior without central coordination. Nature provides stunning examples:

Fireflies in Southeast Asia flash in perfect unison across entire forests. No conductor signals when to flash. Each firefly simply adjusts its timing based on the flashes it sees from neighbors. The result: thousands of fireflies blinking together as one.

Heart pacemaker cells synchronize their electrical pulses to create a unified heartbeat. Each cell is capable of beating on its own, but they listen to their neighbors and adjust, producing the coordinated rhythm that keeps us alive.

Audience applause often transitions from random clapping to rhythmic, synchronized claps. Nobody decides "let's all clap together now" — people naturally adjust to match their neighbors, and global order emerges.

The pattern is consistent: **local adjustment + neighbor awareness = global coordination**. No central controller is needed. Each individual follows simple rules based on local information, and synchronization emerges spontaneously.

Concurrency vs. Synchronization

These terms are related but distinct:

Concurrency means multiple things happening at the same time. Traffic lights are inherently concurrent — all 12 intersections in a grid operate simultaneously, each running its own cycle, making its own decisions. Concurrency is a structural property; it describes *how* the system is organized.

Synchronization means those concurrent entities are coordinating their timing. Without synchronization, 12 concurrent lights would each cycle independently, creating chaotic, scattered traffic. With synchronization, they align their phases, creating smooth green waves.

An analogy: An orchestra has 50 musicians playing concurrently (at the same time). But if they're not synchronized (same tempo, coordinated timing), the result is noise, not music. Concurrency is necessary but not sufficient — we need synchronization to achieve harmony.

Part 2: The Kuramoto Model

How Oscillators Synchronize

In 1975, Japanese physicist Yoshiki Kuramoto proposed a mathematical model explaining how coupled oscillators synchronize. An oscillator is anything that cycles repeatedly: a pendulum, a heartbeat, a blinking firefly, or — in our case — a traffic light cycling between green phases.

Each oscillator has a **phase** (where it is in its cycle) and a **natural frequency** (how fast it wants to cycle). The key insight is the coupling: each oscillator adjusts its speed based on its neighbors' phases.

The Kuramoto coupling formula is:

coupling = $(K/N) \times \Sigma \sin(\phi_j - \phi_i)$

Where:

- K is the coupling strength (how much to listen to neighbors)
- N is the number of neighbors
- ϕ_j is a neighbor's phase
- ϕ_i is my phase

The sine function is crucial. If my neighbor is ahead of me ($\phi_j > \phi_i$), then $\sin(\phi_j - \phi_i)$ is positive, so I speed up. If my neighbor is behind me, I slow down. If we're synchronized ($\phi_j = \phi_i$), then $\sin(0) = 0$, and I don't change — we're in equilibrium.

Why Sine?

Using sine rather than a simple linear difference ($\phi_j - \phi_i$) has important advantages:

- **Bounded**: sin(x) is always between -1 and +1, preventing wild oscillations
- **Circular**: Phase is circular (0 to 2π, then wraps). Sine correctly handles this — $\sin(6.2 - 0.1) \approx \sin(-0.18)$, recognizing these phases are close
- **Smooth equilibrium**: $\sin(0) = 0$ means synchronized oscillators feel no force — they stay synchronized
- **Gradual adjustment**: Near synchronization, sin(small) ≈ small, giving gentle corrections

The Clock Model

I visualize each intersection as a clock. The phase (ϕ) is the angle of the clock hand, ranging from 0 to 2π radians. The hand spins counterclockwise at a rate determined by the frequency.

The phase maps to traffic signals:

- **Phase 0 to π**: North-South gets green
- **Phase π to 2π**: East-West gets green

When all intersection clocks are aligned (same phase), a car traveling through the corridor encounters green after green — the green wave emerges.

Part 3: The Surtrac System

Beyond Pure Synchronization

Kuramoto alone has a limitation: it synchronizes oscillators but doesn't care *where* they synchronize or *what* the traffic actually needs. All intersections might lock together at phase π/2, which is fine mathematically but ignores that North-South traffic might be much heavier than East-West.

Surtrac (Scalable Urban Traffic Control), developed at Carnegie Mellon University, addresses this by enabling intersections to:

- **Communicate** predictions to neighbors ("I'll send you 5 cars in 10 seconds")
- **Adapt** their timing based on actual traffic demand

The Prediction System

Each intersection generates predictions for its neighbors. If I have 6 cars in my South queue (wanting to go North), I predict that about half (3 cars) will go straight to my North neighbor. I package this into a message:

```
Prediction {
  expected_cars: 3,
  release_time: current_time + time_until_green,
  my_phase: 1.2
}
```

I send this to my North neighbor, who uses it to anticipate incoming traffic.

Note: In our simulation, predictions become reality — if I predict 3 cars, my neighbor receives exactly 3 cars. This is a simplification. Real Surtrac systems have sensors to verify actual arrivals and correct prediction errors. We accept this simplification to focus on the core concepts.

Demand-Based Adjustment

Surtrac contributes two adjustments to each intersection's behavior:

Frequency boost: More cars waiting means faster cycling. If total demand is high, the intersection speeds up its cycle to serve more cars per minute.

freq = BASE_FREQUENCY × (1 + 0.2 × demand_ratio)

Direction nudge: If North-South has more cars than East-West, the phase is nudged toward 0 (NS green). This biases the light toward the busier direction without breaking synchronization.

nudge = 0.05 × (0.5 - ns_ratio)

The Complete Phase Update

Combining Kuramoto and Surtrac, each intersection updates its phase every time step:

$\Delta\phi = (\omega + \text{coupling}) \times \Delta t + \text{nudge}$

Where:

- ω is the natural frequency (Surtrac: based on local demand)
- coupling is the Kuramoto synchronization force (based on neighbor phases)
- nudge is the direction bias (Surtrac: favor busier direction)
- Δt is the time step (0.5 seconds)

This elegant equation balances global coordination (Kuramoto) with local responsiveness (Surtrac).

Part 4: The Threading Model

Why Threads?

The simulation models 12 intersections in a 3×4 grid. Each intersection operates independently and simultaneously — a natural fit for concurrent programming with threads.

Each intersection becomes one thread, plus one main thread for coordination and visualization — 13 threads total.

Why not use a simple for-loop? We could, and it would be simpler:

```
for each tick:
  for i = 0 to 11:
    update_intersection(i)
```

But this would be *sequential*, not concurrent. In the real world, traffic lights don't take turns — they all operate at the same time. Using threads models this reality and, more importantly, teaches crucial concepts about concurrent programming: shared data, race conditions, and synchronization primitives.

The Shared Data Problem

Threads share memory. This is both powerful and dangerous.

Each intersection has:

- **queue[4]**: Cars waiting from each direction (private — only this thread accesses it)
- **outgoing[4]**: Predictions I'm sending (private — only I write it)
- **incoming[4]**: Predictions from neighbors (SHARED — neighbors write, I read)

The problem: When I5 reads its incoming[] array to calculate demand, threads I1, I4, I6, and I9 might be writing to it simultaneously. Without protection, this causes **race conditions** — one thread reads while another writes, resulting in corrupted or inconsistent data.

Mutex: The Bathroom Lock

A **mutex** (mutual exclusion) ensures only one thread accesses shared data at a time. It's like a bathroom lock: you lock before entering, do your business, then unlock. Others wait outside until you're done.

```
pthread_mutex_lock(&neighbor->msg_lock);
neighbor->incoming[recv_dir] = my_outgoing[d];
```

```
pthread_mutex_unlock(&neighbor->msg_lock);
```

Critical detail: We lock the *neighbor's* mutex, not our own. We're writing to the neighbor's data, so we need to protect the neighbor's data. Locking our own mutex would be like locking our own bathroom while using the neighbor's — it doesn't protect anything useful.

Without mutex, we get **torn writes**: Thread I4 writes half of a Prediction struct, Thread I6 writes the other half, and I5 reads garbage — a mix of two different predictions.

Barrier: The Checkpoint

A **barrier** ensures all threads reach a certain point before any proceed. It's like a hiking group rule: everyone waits at the checkpoint until the last person arrives, then everyone continues together.

Our simulation has three phases per tick:

- **Communicate**: All threads generate and send predictions
- **Update**: All threads read predictions and update their phase
- **Traffic**: All threads simulate car arrivals and departures

The barrier between Phase 1 and Phase 2 is critical. We must ensure all predictions are written before any thread starts reading. Otherwise, Thread I0 might read I2's incoming[] before I2 finished writing — getting stale data from the previous tick.

```
pthread_barrier_wait(&tick_barrier);  // All 13 threads must arrive
```

Mutex vs. Barrier: Different Problems

Mutex prevents simultaneous access to the *same* data. It protects data integrity during the overlap when multiple threads might touch the same memory location.

Barrier enforces ordering between *phases*. It ensures all of Phase 1 completes before any of Phase 2 begins.

We need both:

- Mutex alone: Phase 1 writes are safe, but a fast thread might start Phase 2 while slow threads are still in Phase 1
- Barrier alone: Phases are ordered, but during Phase 1, two threads might write to the same incoming[] simultaneously

Together, they provide both data integrity and phase ordering.

Part 5: What Happens Under the Hood

Inside a Mutex

At the hardware level, mutex relies on **atomic instructions** like Compare-And-Swap (CAS):

```
CAS(address, expected, new_value):
    if *address == expected:
        *address = new_value
        return true
    else:
        return false
```

This happens in a single, indivisible CPU operation. Even if two threads call CAS at the exact same nanosecond, hardware guarantees only one succeeds.

Inside a Barrier

A barrier is implemented using a mutex and a **condition variable**:

```
barrier_t {
    mutex_t mutex;
    cond_t cond;
    int count;        // Threads currently waiting
    int threshold;    // Total threads needed
    int generation;   // Distinguishes consecutive barriers
}
```

When a thread arrives:

- Lock the mutex
- Increment count
- If count < threshold: wait on condition variable (releases mutex, sleeps)
- If count == threshold: reset count, increment generation, broadcast to wake all waiters
- Unlock mutex

The **generation** counter prevents a subtle bug: without it, a fast thread reaching the *next* barrier could be confused by the wakeup signal from the *previous* barrier.

Condition Variables

A **condition variable** solves the "wait for condition" problem. Unlike a mutex (which just provides exclusion), a condition variable lets a thread sleep until some condition becomes true.

```
pthread_cond_wait(&cond, &mutex);
```

This atomically:

- Releases the mutex
- Adds the thread to the wait queue
- Sleeps

When another thread calls pthread_cond_broadcast(&cond), all waiting threads wake up and re-acquire the mutex before returning from cond_wait.

Without condition variables, we'd need busy-waiting loops that waste CPU.

Part 6: From Chaos to Order

The Simulation

The simulation starts with chaos: 12 intersections with random phases between 0 and 2π. Their clock hands point in all directions. Coherence (a measure of synchronization) is around 0.2 — nearly random.

Each tick (0.5 seconds of simulated time):

- **Communicate**: Each intersection predicts future traffic and sends messages to neighbors
- **Update**: Each intersection reads neighbor data, calculates Kuramoto coupling and Surtrac adjustment, and updates its phase

- **Traffic**: Random cars arrive, predicted cars from neighbors arrive, cars depart if their light is green

Emergence

Within 30-60 seconds of simulated time, something remarkable happens. The random phases converge. Clock hands align. Coherence rises toward 1.0.

No intersection was *told* to synchronize. Each simply followed local rules:

- Speed up if neighbors are ahead
- Slow down if neighbors are behind
- Cycle faster if traffic is heavy
- Favor the busier direction

Global order emerged from local interactions. This is emergence — the "magic" of synchronization.

The Green Wave

Once synchronized, a car entering the corridor at the right moment experiences green after green. The wave of cars released from one intersection arrives at the next just as it turns green. Traffic flows smoothly. Average wait times drop. The system *works*.

Part 7: Key Insights

Local Rules, Global Order

The most profound insight: complex global behavior can emerge from simple local rules. No central traffic controller optimizes the system. Each intersection knows only about itself and its immediate neighbors. Yet coordination emerges network-wide.

This principle appears throughout computer science: distributed systems, swarm intelligence, neural networks. Local computation can produce global patterns.

Concurrency Requires Coordination

Running 12 threads simultaneously is easy. Making them work together correctly is hard. Shared data creates race conditions. Unsynchronized phases create stale reads. Without mutex and barrier, the simulation would produce garbage.

This is the fundamental challenge of concurrent programming: independence is easy, coordination is hard.

The Right Abstraction Matters

Viewing traffic lights as oscillators (Kuramoto) transforms an engineering problem into a physics problem with well-understood solutions. Adding communication (Surtrac) makes the physics practical. The combination is more powerful than either alone.

Good abstractions reveal hidden structure. The clock model — phase as an angle, frequency as rotation speed — makes the math intuitive and the code clear.

Simplifications Have Limits

Our model simplifies reality:

- Predictions are deterministic (real systems need sensors and correction)
- Only two phases per light (real intersections have yellow lights, pedestrian phases, turn arrows)
- Grid topology (real networks are irregular)
- No broken lights or crashed threads (real systems need fault tolerance)

These simplifications are appropriate for a conceptual demonstration but would need addressing in production software.

Conclusion

What began as an observation at a crosswalk became an exploration of synchronization, concurrency, physics, and traffic engineering.

Traffic lights are concurrent: they operate independently and simultaneously. Making them synchronize — align their timing without central control — requires both a mechanism for local coordination (Kuramoto coupling) and a way to adapt to actual traffic (Surtrac predictions).

Implementing this as a threaded simulation taught me core concepts of concurrent programming: threads share memory, shared memory creates race conditions, mutex protects data integrity, barriers enforce phase ordering, and condition variables enable efficient waiting.

The result is a system where 12 simple rules-following intersections produce emergent global coordination. Cars flow in waves. Green waves emerge. The whole becomes greater than the sum of its parts.

That's the magic of sync.

Implementation: C with POSIX Threads (pthreads). 12 intersection threads + 1 main thread. Mutex protects incoming[] arrays. Barrier synchronizes simulation phases. Kuramoto coupling constant K=0.3. Surtrac nudge coefficient 0.05. Time step Δt=0.5 seconds. 60-second default cycle.

References

- Kuramoto, Y. (1975). "Self-entrainment of a population of coupled non-linear oscillators." International Symposium on Mathematical Problems in Theoretical Physics.
- Smith, S.F., et al. (2013). "Surtrac: Scalable Urban Traffic Control." Transportation Research Board Annual Meeting.
- Strogatz, S.H. (2000). "From Kuramoto to Crawford: Exploring the onset of synchronization in populations of coupled oscillators." Physica D.