Universidad del Valle de Guatemala

Facultad de ingeniería

Computación Paralela y Distribuida

Catedrático: Luis García

Mini mini Proyecto Tráfico

Nelson Eduardo García Bravatti 22434

Guatemala, agosto de 2025

Breve descripción del problema.

Se modela una intersección con 4 carriles y un semáforo por carril. Un conjunto de vehículos se aproxima a la línea de alto con velocidad constante. Cada semáforo alterna entre verde (V), amarillo (A) y rojo (R) con duraciones distintas por semáforo (todas ≤ 10 s). Un vehículo:

- Avanza si no está esperando,
- Se detiene a ~2 m de la línea si encuentra rojo,
- Cruza una sola vez cuando alcanza la línea con V/A y termina su participación.

La simulación termina cuando todos los vehículos hayan cruzado. Se imprimen "snapshots" periódicos (cada k segundos simulados) y, al final, métricas como vehículos que cruzaron, espera promedio, tiempo simulado y tiempo de ejecución (wall-clock).

Estrategia de paralelización utilizada (qué se paralelizó, por qué y cómo).

Qué se paralelizó:

Movimiento de vehículos: es el trabajo dominante en cada iteración. Cada vehículo se actualiza de manera casi independiente (solo consulta el estado del semáforo de su carril), lo que lo hace ideal para paralelizar.

Conteo de cruces por iteración: se realiza con una reducción sobre un contador entero (crossed_step).

Qué no se paralelizó y por qué:

Actualización de semáforos: son solo 4 elementos por defecto → paralelizar ese bucle genera más overhead que beneficio. Se ejecuta en un único hilo (#pragma omp single).

Limpieza de eventos por iteración: se realiza en el mismo bloque single. Podría paralelizarse, pero el beneficio suele ser marginal frente al coste de sincronización.

Cómo se implementó:

Región paralela persistente:

```
#pragma omp parallel default(shared)
{
   for (;;) {
      #pragma omp single
      { /* actualizar semáforos + limpiar eventos + reiniciar crossed_step */ }

      #pragma omp for schedule(static) reduction(+:crossed_step)
      for (int i = 0; i < num_vehicles; ++i) {
         // mover vehículo i
      }

      #pragma omp single
      { /* acumular totales, avanzar tiempo, imprimir snapshot si aplica */ }

      // condición de salida sincronizada
   }
}
```

Motivo: evitar el coste de crear/destruir equipos de hilos en cada iteración (fork-join overhead). Con una sola región, los hilos permanecen activos y el coste amortizado cae drásticamente.

Trabajo por hilos en el lazo de vehículos:

- schedule(static): asigna bloques contiguos de vehículos a cada hilo → bajo overhead y mejor localidad de caché cuando los accesos son uniformes.
- reduction(+:crossed_step): suma segura y eficiente de los cruces detectados por cada hilo en la iteración.
- Escritura en crossed_now[i]: cada hilo escribe en índices únicos → evita condiciones de carrera sin necesidad de locks.

Sincronización mínima:

- single para tareas pequeñas y actualización de totales (una sola escritura global por iteración).
- Barreras implícitas en for y single (y dos barreras ligeras al inicio del ciclo) para consistencia sin penalizar de más.

Justificación del uso de OpenMP (paralelismo dinámico, anidado, etc.).

Por qué OpenMP para este problema:

- Modelo de memoria compartida natural: semáforos y arreglo de vehículos viven en el mismo proceso; los hilos comparten el estado sin IPC.
- Intrusividad baja: se añaden pragmas a C existente; fácil de mantener y leer para un laboratorio.
- Portabilidad: compila con gcc/clang/icx en Linux/macOS/Windows.

Decisiones sobre características específicas:

Tamaño de equipo y hilos dinámicos

- Se usa omp_set_dynamic(0) (dinámico desactivado) y un equipo estable.
- Justificación: cambiar el tamaño del equipo por iteración introduce overhead y variabilidad (especialmente con N=100–200). Con carga homogénea por iteración, un equipo fijo + schedule(static) es más eficiente y predecible.

Paralelismo anidado (omp_set_nested)

- No se usa: el patrón del algoritmo es un único bucle externo con trabajo masivo por iteración. Anidar paralelismo complicaría la sincronización, aumentaría el overhead y no aporta beneficio real dado el tamaño del problema.

Planificación (scheduling)

- schedule(static) por defecto.
- Justificación: la carga por vehículo por iteración es bastante uniforme; static minimiza el coste del scheduler.
- Alternativa: schedule(guided) solo si observas alto desbalance (p. ej., si muchos vehículos terminan pronto en un hilo y quedan pocos en otros). En tus pruebas “clásicas” no suele ser necesario.

Región paralela persistente vs múltiples regiones

- Se eligió persistente para eliminar el coste de fork-join por iteración, crucial cuando cada paso tiene poco trabajo relativo (actualización simple + restas de posición).

I/O y medición de tiempo

- El I/O de consola es serial y domina con intervalos de impresión pequeños. Para medir rendimiento, sube print_every (p. ej., 5–10) o desactiva prints.
- Tiempos con omp_get_wtime() y 6 decimales para mayor precisión en el wall-clock.

Detalles del modelo y estructuras:

- TrafficLight: estado (RED/GREEN/YELLOW), time_in_state, y duraciones distintas por semáforo (t_green, t_yellow, t_red, todas ≤ 10 s).
- Vehicle: pos, speed, waiting, finished (cruza una vez), total_wait, crossings.
- Intersection: arreglo de semáforos y parámetros de detención.
- Ciclo (por iteración): actualizar semáforos → mover vehículos → acumular cruces → snapshot opcional.
- Criterio de término: total_crossed == num_vehicles.