HPC Tutorial: Synchronization Strategies in N-Body Tree Codes

NOTE: *This text has been generated by the Gemini 3.0 pro LLM, under my instructions (see the complete prompt at the end). I slightly edited the text to fix some inaccuracies.*

Introduction

In N-Body simulations (like astrophysical tree codes), particles often drift at different timesteps. Before a thread can calculate the force on a particle, it must ensure that particle has been drifted to the current time `update_time`.

We are simulating a scenario where:

1. We have `N` particles, of which only a subset is active.
2. Multiple threads concur in “using” the particles.
3. If a particle is "old", it must be updated before being used.
4. **Constraint:** Only one thread can update a specific particle at a time (Race Condition).

The rationale is that every thread may have to calculate the Gravitational force on its own “active” particles. Those particles interact with their close neighbours only (the more diatnt particles are treated using the tree); however, every neighbour must be drifted to its current position before the Gravity between the two particles is calculated.

We will explore three evolution stages of solving this problem using OpenMP.

Version 1: The Coarse-Grained Approach (`01_critical`)

The simplest way to prevent data races is to use a **Critical Section**. This is the "One Bathroom Key for the Whole Building" approach.

The Implementation

```
if ( P[idx].time < update_time )
{
  #pragma omp critical
  {
      // Double-check logic should be here!
      update_particle( &P[idx], update_time );
  }
}
```

The Memory Model

- **Entry:** When entering a critical region, OpenMP ensures all previous memory operations are visible (Implicit Flush/Acquire).
- **Exit:** When leaving, OpenMP ensures the changes (the new time) are flushed to main memory (Implicit Flush/Release).

Performance Analysis

- **Pros:** Very easy to write. Correctness is (almost) guaranteed by the compiler.
- **Cons: Massive Serialization.** If Thread 0 is updating Particle A, Thread 1 cannot update Particle B. It must wait. The critical section is a global lock.
- **Didactic Note:** This approach scales poorly as the number of threads increases.

Version 2: Fine-Grained Locking (`02_omplock`)

To improve parallelism, we move the lock from the global scope to the data scope. This is the "Key for Every Room" approach. We add an `omp_lock_t` to every particle.

The Implementation

We use `omp_test_lock`. This is non-blocking. If we can't get the lock, we don't just sleep; we check *why* the lock is held (is someone else updating it?).

```
// Try to grab the specific lock for THIS particle
int got_it = omp_test_lock( &P[idx].lock );

if ( got_it ) {
    // We own the particle. Update it.
    update_particle( &P[idx], update_time );
    omp_unset_lock( &P[idx].lock );
} else {
    // Someone else has the lock.
    // We don't need to acquire it; we just need to wait
    // for them to finish the work.
    while ( P[idx].time < update_time ) {
        nanosleep(...); // Spin/Wait
    }
}
```

The Memory Model

- `omp_set_lock` / `omp_unset_lock` imply memory fences.
- **Overhead:** `omp_lock_t` is an opaque structure. It can be large (eating up memory bandwidth) and the function calls to the OpenMP runtime library add overhead (function pointer jumps, system calls).

Version 3: Custom Atomic Flags (`03_lock`)

This is the "High-Performance/Lightweight" approach. We strip away the heavy OpenMP lock API and implement a custom spinlock using standard integer variables and atomic operations.

The Concept: Atomic Capture

We use an `int lock` inside the particle struct.

- `0`: Unlocked.
- `>0`: Locked.

We need an operation that reads the value AND modifies it in one go (atomic).

```
int _lock_;
#pragma omp atomic capture
{ _lock_ = P[idx].lock; P[idx].lock++; }
```

This is equivalent to `fetch-and-add`. If `_lock_` was 0, we successfully transitioned it to 1. We are the owner.

The Memory Model: Acquire & Release

This version requires us to manually handle memory consistency. We cannot assume other threads see our changes immediately without these clauses.

1. **Acquire:** When we read the time, we use `#pragma omp atomic read acquire`. This ensures we don't read a cached/stale value from before the update.
2. **Release:** When the updater finishes, they perform `#pragma omp write release` on the `time` and `lock`. This ensures that **all** calculations done inside the function are flushed to memory *before* the lock is set to 0.

Code Snippet

```
if ( _lock_ == 0 ) {
    // Update
    update_particle( ... );

    // RELEASE: Ensure update is visible before unlocking
    #pragma omp atomic write release
    P[idx].lock = 0;
}
```

Summary of Trade-offs

Version	Complexity	Memory Overhead	Concurrency	Latency
Critical	Low	Zero	Very Low (Global blocking)	High (Waiting for unrelated threads)
OMP Lock	Medium	High (`omp_lock_t` size)	High (Per-particle)	Medium (Library overhead)
Atomic	High	Low (`int`)	High (Per-particle)	Low (CPU instruction level)

Possible False Sharing

Let me (the Prof.) add this consideration.
In all versions (especially 3), notice that `part_t` is small (12-16 bytes).

- Cache lines are typically 64 bytes.
- `P[0]`, `P[1]`, `P[2]` likely sit on the *same* cache line.
- If Thread A writes to `P[0]` (locking it), the hardware invalidates the *entire cache line*.
- Thread B trying to read `P[1]` misses the cache, even though logically they are different particles.
- **Solution:** Pad the structure to 64 bytes to ensure 1 particle = 1 cache line.

However, this comes with a much larger memory imprint, which may be a serious cons. It is up to you to judge, either theoretically or by measuring, case by case, which is the best strategy.

The complete prompt

I have uploaded 3 codes, numbered from 01 to 03.
In these codes there are 3 different implementations of a solution for the following problem. I mimic what happens in a N-body Tree code, where particles have individual time-steps and, above all, they do not interact directly with all other particles. In fact, in a Tree code, the gravitational interaction beyond a given distance is calculated using multi-polar expansion of distant particles, instead of by pair-wise summation.

In this code, a number N of particles is processed in parallel by openmp threads, each threads chooses randomly how many particles it should actually process. That number if called myN.
The fact that the total number of processed particles is smaller than N replicates the fact that not all the particles are active at every timestep.

However, to process a particle, which is done in the function use_particle(), that same particle must be "updated". The update amounts to set the particle's time to the present time.

So every thread has to check whether a particle is updated before using it; if that is not the case, it has to update the particle.
However, since many threads may have the same particle in their list to be "used", this update must be protected from data race.

The three different versions deal with this aspect differently.
Version 01 does that by using a critical region.
Version 02 uses omp locks.
Version 03 uses signals among threads, with a more lightweight approach.

My aim is to guide a class of HPC student through this, explaining the changes from 01 to 02 and then to 03, all the inefficiencies (for instance, related to the implicit flushes at entry and exit of the critical and of the OpenMP lock) in the frame of the OpenMP memory model.