Parallel Heap Implementation using Double-Linked Lists: A Journey Through Synchronization and Memory Models

- This text has been produce by Claude Sonnet 4.5, following this prompt:
  «In this folder I have collected 6 codes: the names follows the convention
  `$DD_linked_list.$description.c`.
  The aim is to illustrate to an HPC class how to synchronize threads and some subtletiy on the memory model.
  The objective is implementing a heap by using a double-linked list.
  Code 01 is a serial implementation.
  Code 02 is a very minimal openmp parallelization, very inefficient because there is a lot of contention on the locks.
  Version 03 is more efficient, probably the best compromise between efficiency and code complexity.
  Codes 04 and 05 illustrates a strategy where there is a local walk to repair broken links; however, both of them have
  bugs and inaccuracies.
  Version 06 if the final correct version, that also correctly deals with the openmp memory model.
  I want that you write a digression the brings the student across all the versions, pinpointing strategies, discussing the inefficiency in 03, the bugs in 04 and 05.»

Introduction

This tutorial presents a progressive implementation of a parallel heap data structure using a sorted double-linked list. We'll examine six versions of the code, each building upon the previous one, introducing new parallelization strategies, uncovering subtle bugs, and finally arriving at a correct implementation that properly handles OpenMP's memory model.

The goal is to understand:

1. Different synchronization strategies in parallel programming
2. The performance trade-offs between coarse-grained and fine-grained locking
3. Deadlock scenarios and how to avoid them
4. Memory model considerations in OpenMP
5. Common pitfalls in task-based parallelism

Version 01: Serial Implementation (`01_linked_list.serial.c`)

Overview

The baseline implementation provides a serial sorted double-linked list with:

- `find()`: Searches for the insertion point for a given value
- `find_and_insert()`: Inserts a new node in sorted order

- `walk()`: Verifies the list is correctly sorted
- `get_head()`: Finds the head of the list

Key Characteristics

- **No concurrency**: Pure serial execution
- **Bidirectional traversal**: Can walk forward or backward depending on the starting point
- **Simple pointer manipulation**: Direct reads and writes to `next` and `prev` pointers

Code Structure

```c
typedef struct llnode {
    int data;
    struct llnode *next;
    struct llnode *prev;
} llnode_t;
```

The `find()` function walks the list without any protection, and `find_and_insert()` directly updates the pointers.

Performance Baseline

This version serves as our performance baseline. All parallel versions should aim to improve upon this while maintaining correctness.

Version 02: Minimal Parallelization (`02_linked_list.minimal.c`)

Overview

The first parallel version introduces OpenMP tasks with the simplest possible synchronization strategy.

Key Changes

1. **Global lock**: A single `insertion_lock` protects all insertions
2. **OpenMP tasks**: Work is distributed using `#pragma omp task`
3. **Atomic operations**: Pointers are declared as `_Atomic` and accessed with `atomic_load_explicit()` and `atomic_store_explicit()`

Synchronization Strategy

```
omp_lock_t insertion_lock;

#pragma omp task firstprivate(n, head, this_batch_size)
{
    for (int batch = 0; batch < this_batch_size; batch++) {
        int new_value = nrand48(seeds) % norm;
        omp_set_lock(&insertion_lock);
        find_and_insert(head, new_value, &head);
        omp_unset_lock(&insertion_lock);
    }
}
```

The Problem: Excessive Lock Contention

This is the most inefficient parallel implementation.

Every insertion, regardless of where it occurs in the list, must acquire the same global lock. This means:

- **Serialization**: Only one thread can insert at any time
- **No parallelism**: Despite using multiple threads, insertions are completely serialized
- **Cache coherence overhead**: All threads contend for the same lock variable
- **Poor scalability**: Performance may be worse than serial due to synchronization overhead

Performance Analysis

With N threads:

- **Theoretical speedup**: None (completely serialized)
- **Actual performance**: Likely slower than serial due to:
    - Task creation overhead
    - Lock acquisition/release overhead
    - False sharing on the lock variable

Why Atomic Operations?

Even though we have a global lock, we use atomic operations for pointer accesses. This is a **bridge to the next version** and ensures that:

- Other threads performing unlocked reads (like in `get_head()` or `walk()`) don't encounter undefined behavior
- We're prepared for finer-grained locking strategies

Version 03: Fine-Grained Locking (`03_linked_list.step1.c`)

Overview

This version introduces **per-node locking**, dramatically improving parallelism by allowing concurrent insertions at different locations in the list.

Key Changes

1. **Per-node locks**: Each node has its own `omp_lock_t`
2. **Optimistic search**: `find()` runs without locks, giving a hint
3. **Lock acquisition protocol**: Acquires locks on neighbor nodes
4. **Validation**: Checks if insertion point is still valid after locking

Data Structure

```
typedef struct llnode {
    int data;
    omp_lock_t lock;
    _Atomic(struct llnode *)next;
    _Atomic(struct llnode *)prev;
} llnode_t;
```

The Algorithm: Optimistic Concurrency

Step 1: Optimistic Search (No Locks)

```
llnode_t *prev = NULL, *next = NULL;
find(start, value, &prev, &next);
// Now: prev->data < value <= next->data (hopefully!)
```

The `find()` function runs **without holding any locks**. This means:

- Multiple threads can search simultaneously
- The result might be stale by the time we acquire locks
- Other threads might have inserted nodes between `prev` and `next`

Step 2: Deadlock-Free Lock Acquisition

```
int locks_acquired = 0;
while (!locks_acquired) {
    // Try to acquire prev first
    if (prev != NULL) {
        while (omp_test_lock(&(prev->lock)) == 0) {
            if (use_taskyield) {
                #pragma omp taskyield
            }
```

```c
        }
        locks_acquired = 1;
    }

    // Then try to acquire next
    if (next != NULL) {
        locks_acquired = my_test_lock(next, me);
        if (!locks_acquired) {
            // Failed to get next; release prev and retry
            if (prev != NULL) {
                my_unset_lock(prev);
            }
            if (use_taskyield) {
                #pragma omp taskyield
            }
        }
    }
}
```

Why is this deadlock-free?

- Always acquires locks in the same order: `prev` first, then `next`
- If can't get both, releases all and retries
- Never holds a lock while waiting for another (no circular wait)

Step 3: Validation

```c
llnode_t *prev_next = atomic_load_explicit(&prev->next, memory_order_relaxed);
llnode_t *next_prev = atomic_load_explicit(&next->prev, memory_order_relaxed);

if ((prev != NULL && prev_next != next) ||
    (next != NULL && next_prev != prev)) {
    // Someone inserted between prev and next!
    // Count the clash and retry
    atomic_fetch_add_explicit(clashes, 1, memory_order_relaxed);

    // Release locks
    if (prev != NULL) my_unset_lock(prev);
    if (next != NULL) my_unset_lock(next);

    continue;  // Go back to Step 1
}
```

If validation fails, we **restart the entire process from Step 1**, calling `find()` again from the original `head`.

Step 4: Insertion

If validation passes, we insert the new node and release locks.

Performance Characteristics

Advantages:

- ✅ Multiple threads can insert at different locations simultaneously
- ✅ Deadlock-free
- ✅ Correct (maintains sorted order)
- ✅ Good parallelism when insertions are spread across the list

Disadvantages:

- ❌ When validation fails, **restarts from the original head**
- ❌ In high-contention scenarios, threads repeatedly find the same insertion point
- ❌ Wasted work: the optimistic search is repeated even when we already have a locked node close to the insertion point

The Inefficiency

Consider this scenario with high contention:

1. Thread A searches from head, finds insertion point between nodes X and Y
2. Thread A acquires locks on X and Y
3. Thread B inserted a node between X and Y while A was searching
4. Thread A's validation fails
5. **Thread A releases both locks and searches again from head**

The inefficiency is in step 5: Why restart from `head` when we already have information about where the insertion point is? We could walk locally from X or Y to find the correct insertion point while keeping at least one lock.

This is exactly what versions 04 and 05 attempt to do.

Version 04: Local Walk with Deadlock Bug (`04_linked_list.walk_local_with_deadlock.c`)

Overview

This version attempts to improve upon Version 03 by implementing a **local repair walk**: when validation fails, instead of restarting from the head, we walk locally from the locked nodes to find the correct insertion point.

Key Changes

When validation fails, the code now performs a **repair walk**:

```
if ((prev != NULL) && (prev->next != next)) {
    // The next pointer has changed
    // Walk forward from prev to find the new next
```

```
    if (next != NULL) {
        omp_unset_lock(&(next->lock));  // Release old next
    }

    next = prev->next;
    while (next) {
        omp_set_lock(&(next->lock));  // ⚠️ BLOCKING CALL

        if (next->data >= value)
            break;  // Found correct next

        omp_unset_lock(&(prev->lock));
        prev = next;
        next = next->next;
    }
}
```

The Deadlock Bug

This implementation has a **critical deadlock vulnerability** at line 340 and 374 where `omp_set_lock()` is used during the repair walk.

Deadlock Scenario

Consider two threads trying to insert values:

- Thread A wants to insert value 50, has lock on node(40), walking forward
- Thread B wants to insert value 60, has lock on node(70), walking backward

Timeline:

```
T1: Thread A locks node(40), validation fails, starts walking forward
T2: Thread B locks node(70), validation fails, starts walking backward
T3: Thread A needs to lock node(50) to continue forward
T4: Thread B needs to lock node(60) to continue backward
T5: Thread A reaches node(60), calls omp_set_lock(&node(60)->lock)  // BLOCKS
T6: Thread B reaches node(50), calls omp_set_lock(&node(50)->lock)  // BLOCKS
```

Deadlock state:

- Thread A holds lock on node(40) or node(50), waiting for node(60)
- Thread B holds lock on node(70) or node(60), waiting for node(50)
- Circular wait: A → B → A

Why is This Different from the Initial Lock Acquisition?

In the initial lock acquisition (Step 2), the code uses:

```c
locks_acquired = my_test_lock(next, me);
if (!locks_acquired) {
    // Release prev and retry
    my_unset_lock(prev);
}
```

This uses `omp_test_lock()` which is **non-blocking**: if the lock isn't available, it returns immediately, and we can release our held locks and retry.

But in the repair walk, the code uses `omp_set_lock()` which is **blocking**: if the lock isn't available, the thread waits indefinitely, still holding other locks.

Summary of Version 04

- ✅ Good idea: local repair walk reduces wasted work
- ❌ **Fatal flaw**: deadlock due to blocking lock acquisition during repair
- ❌ Not production-ready

Version 05: Local Walk with Firstprivate Bug (`05_linked_list.walk_local_with_firstprivate_bug.c`)

Overview

This version attempts to fix the deadlock by using `my_test_lock()` with limited attempts, but introduces a **different bug related to OpenMP task data sharing**.

Changes from Version 04

Fix for Deadlock

```c
int attempts = 0;
while (++attempts < MAX_ATTEMPTS) {
    int got_next_lock = my_test_lock(next, me);
    if ((!got_next_lock) && (use_taskyield)) {
        #pragma omp taskyield
    }
}

if (attempts == MAX_ATTEMPTS) {
    // Too many attempts; release everything and restart
    if (prev != NULL) { start = prev; my_unset_lock(prev); }
    if (next != NULL) { start = next; my_unset_lock(next); }
    continue;  // Restart from new start point
}
```

The deadlock is avoided by:

- Using non-blocking `my_test_lock()` instead of blocking `omp_set_lock()`
- Limiting attempts with `MAX_ATTEMPTS`
- If we can't acquire the lock after many attempts, release everything and restart

This is a valid approach!

The Firstprivate Bug

The **real bug** in this version is subtle and appears in the task creation code at **line 621**:

```
llnode_t *start = head;  // Line 607: shared variable

#pragma omp task  //  BUG: missing firstprivate(start, head)
{
    for (int batch = 0; batch < this_batch_size; batch++) {
        int new_value = lrand48() % norm;
        find_and_insert_parallel(start, new_value, mode, &clashes, &start);
        //                       ^^^^^                               ^^^^^^
        //                       reads start                    writes to start
    }
}
```

The Problem: Data Race on `start`

1. **`start` is declared outside the task** (line 607)
2. **The task does NOT have `firstprivate(start)`**
3. **All tasks share the same `start` variable**
4. **Each task updates `start` through the `&start` parameter** in `find_and_insert_parallel()`

This creates a **data race**:

- Multiple tasks read and write the same `start` pointer concurrently
- No synchronization protects these accesses
- Tasks interfere with each other's starting point for searches

Consequences

```
Thread 1's Task:
  - Starts with start = node(100)
  - Inserts value, updates start = node(150)

Thread 2's Task (simultaneously):
  - Starts with start = node(200)
  - Reads start → sees node(150) (Thread 1's update!)
  - Incorrect starting point for search
```

This causes:

- **Incorrect behavior**: Tasks search from wrong starting points

- **Potential lost nodes**: Nodes might not be inserted correctly
- **Performance degradation**: Threads stepping on each other's feet

What Should Happen

In Version 06, the task is correctly declared as:

```
#pragma omp task firstprivate(n, head, this_batch_size)
```

With `firstprivate(head)`:

- Each task gets its **own private copy** of `head`
- Updates to `head` inside the task don't affect other tasks
- No data race

Summary of Version 05

- ✅ Fixes the deadlock from Version 04
- ✅ Good local repair walk strategy
- ❌ **Data race on shared variable** due to missing `firstprivate`
- ❌ Can cause incorrect results and lost nodes

Version 06: Correct Implementation (`06_linked_list.walk_local_correct.c`)

Overview

This is the **final, correct version** that combines all the good ideas from previous versions while fixing all the bugs. It properly handles:

- Deadlock-free fine-grained locking
- Efficient local repair walks
- Proper task data privatization
- OpenMP memory model compliance

Key Fixes

1. Fixed Task Declaration (Lines 689)

```
#pragma omp task firstprivate(n, head, this_batch_size)
{
    for (int batch = 0; batch < this_batch_size; batch++) {
        int new_value = nrand48(seeds) % norm;
        find_and_insert_parallel(head, new_value, mode, &clashes, &head);
        //                       ^^^^                                ^^^^
        //                       Each task has its own head pointer
    }
```

```
}
```

Now each task has its own `head` pointer, eliminating the data race.

2. Non-Blocking Local Repair Walk (Lines 426-461)

Forward walk (when `prev` is valid):

```
next = atomic_load_explicit(&prev->next, memory_order_relaxed);

while (next != NULL) {
    // Try to acquire lock with limited attempts
    while (!(got_lock) && (++attempts < MAX_ATTEMPTS))
        got_lock = my_test_lock(next, me);

    if (got_lock) {
        // Successfully locked next
        if (next->data >= value)
            break;  // Found correct insertion point

        // Need to continue walking
        my_unset_lock(prev);
        prev = next;
        next = atomic_load_explicit(&next->next, memory_order_relaxed);
    } else {
        // Too many attempts; force restart
        next = NULL;
    }
}
```

Backward walk (when `next` is valid):

```
prev = atomic_load_explicit(&next->prev, memory_order_relaxed);

while (prev != NULL) {
    // Try to acquire lock with limited attempts
    while ((!got_lock) && (++attempts < MAX_ATTEMPTS))
        got_lock = my_test_lock(prev, me);

    if (got_lock) {
        // Successfully locked prev
        if (prev->data <= value)
            break;  // Found correct insertion point

        // Need to continue walking
        my_unset_lock(next);
        next = prev;
        prev = atomic_load_explicit(&prev->prev, memory_order_relaxed);
    } else {
        // Too many attempts; force restart
        prev = NULL;
    }
```

```
}
```

Key features:

- ✅ Non-blocking: uses `omp_test_lock()` only
- ✅ Limited attempts: avoids infinite loops
- ✅ Graceful degradation: falls back to full restart if repair fails
- ✅ Maintains at least one lock during walk

3. Double Validation (Lines 537-549)

After a successful repair walk, the code validates again:

```
if (got_lock) {
    // Got both locks after repair walk
    // Check AGAIN that insertion point is still valid
    llnode_t *prev_next_check = (prev != NULL) ?
        atomic_load_explicit(&prev->next, memory_order_relaxed) : NULL;
    llnode_t *next_prev_check = (next != NULL) ?
        atomic_load_explicit(&next->prev, memory_order_relaxed) : NULL;

    if (((prev != NULL) && (prev_next_check != next)) ||
        ((next != NULL) && (next_prev_check != prev))) {
        // Someone inserted during our repair walk
        atomic_fetch_add_explicit(clashes, 1, memory_order_relaxed);
        attempts = MAX_ATTEMPTS;  // Force restart
    }
}
```

Why is this necessary?

During the repair walk, we release and reacquire locks. Between releasing `prev` and acquiring the next `prev`, another thread might have inserted. This second validation catches that case.

Memory Model Considerations

This version properly uses C11 atomics with appropriate memory orders:

Relaxed Operations for Pointers

```
atomic_load_explicit(&node->next, memory_order_relaxed)
atomic_store_explicit(&node->next, value, memory_order_relaxed)
```

Why relaxed?

- The locks provide the necessary synchronization barriers
- `omp_set_lock()` has implicit **acquire** semantics
- `omp_unset_lock()` has implicit **release** semantics
- These ensure that all memory operations inside the critical section are properly ordered

The Lock-Based Memory Model

```
Thread A:                          Thread B:
  data = 42;                         omp_set_lock(&lock);
  next->value = data;                   // ↓ acquire barrier
  omp_unset_lock(&lock);                data = next->value;
     ↑ release barrier                  // Sees data = 42
```

The release-acquire pair creates a **happens-before** relationship:

- All writes before `omp_unset_lock()` are visible after `omp_set_lock()`
- No need for stronger memory orders on individual operations

Performance Characteristics

Best-case scenario (low contention):

- Multiple threads insert at different locations simultaneously
- Minimal lock contention
- Near-linear speedup

High-contention scenario:

- Local repair walks reduce wasted work
- Clashes are handled efficiently
- Graceful degradation to full restart when necessary

Compared to Version 03:

- Fewer restarts from head
- More efficient use of already-acquired locks
- Better performance under high contention

Summary of Version 06

- ✅ Deadlock-free
- ✅ Data-race-free (proper `firstprivate`)
- ✅ Efficient local repair walks
- ✅ Double validation for correctness
- ✅ Proper memory model handling
- ✅ Production-ready

Comparative Analysis

Version	Strategy	Efficiency	Correctness	Key Issue
v01	Serial	Baseline	✅ Correct	None

Version	Strategy	Efficiency	Correctness	Key Issue
v02	Global lock	Very poor	✅ Correct	Complete serialization
v03	Per-node locks + global restart	Good	✅ Correct	Inefficient restart from head
v04	Local repair walk	Better	❌ **Deadlock**	Blocking locks in repair walk
v05	Local repair walk + attempt limit	Better	❌ **Data race**	Missing `firstprivate`
v06	All of the above, fixed	Best	✅ Correct	None

Performance Expectations

Assuming N threads and M insertions:

- **v02**: Speedup ≈ 1.0 (possibly < 1.0 due to overhead)
- **v03**: Speedup ≈ 0.5N to 0.8N (depends on contention)
- **v06**: Speedup ≈ 0.7N to 0.95N (better handling of contention)

Key Takeaways

1. Lock Granularity Matters

- **Coarse-grained** (v02): Easy to implement, poor performance
- **Fine-grained** (v03-v06): Complex to implement correctly, much better performance

2. Optimistic Concurrency

- Search without locks (optimistic)
- Validate after locking
- Retry if validation fails

3. Deadlock Prevention

- ✅ **Do**: Use non-blocking locks (`omp_test_lock()`)
- ✅ **Do**: Release all locks before retrying
- ✅ **Do**: Acquire locks in consistent order
- ❌ **Don't**: Block while holding locks (`omp_set_lock()` in v04)

4. Memory Model

- Locks provide synchronization barriers
- Use appropriate memory orders (relaxed is often sufficient with locks)
- Atomic operations prevent undefined behavior, not necessarily races

5. Task Data Sharing

- Always consider what variables tasks share
- Use `firstprivate` for variables each task should have its own copy of
- Remember: default is **shared**, not private!

6. Validation is Critical

- After acquiring locks, check that your assumptions still hold
- Other threads may have modified the structure while you were waiting
- In some cases (v06), validate twice

7. Debugging Concurrent Code

- Use debug output (see `dbgout` macros)
- Track lock ownership (see `SET_OWNER` macros)
- Count and report contentions/clashes
- Consider using ThreadSanitizer (TSan) for race detection

Exercises for Students

1. **Measure performance**: Compile all versions and measure their performance with different numbers of threads and different contention levels. How does the number of clashes in v03 vs v06 compare?
2. **Reproduce the deadlock**: Run v04 with high contention and many threads. Can you reliably trigger the deadlock? How long does it take?
3. **Detect the data race**: Use ThreadSanitizer on v05. What does it report? Does the program still produce correct results sometimes?
4. **Memory orders**: In v06, change some `memory_order_relaxed` to `memory_order_seq_cst`. Does it affect correctness? Performance?
5. **Lock statistics**: Modify v06 to track per-thread statistics:
   - How many times did repair walk succeed?
   - How many times did it fail and trigger a full restart?
   - What's the average length of repair walks?
6. **Alternative strategies**:
   - Implement a version with read-write locks
   - Try a lock-free implementation using compare-and-swap

  - Experiment with different `MAX_ATTEMPTS` values

Conclusion

This journey from serial to correct parallel implementation demonstrates that concurrent programming is subtle and requires careful attention to:

- Synchronization strategies
- Deadlock prevention
- Memory models
- Data sharing semantics

The progression v02 → v03 → v04 → v05 → v06 shows that even experienced programmers can make mistakes, and that correctness must be verified through careful reasoning, testing, and potentially formal methods.

The final version (v06) achieves good performance while maintaining correctness, demonstrating that efficient parallel data structures are achievable with proper design and implementation.