Why Shuo's Pre-Loading Solution Actually Works

The Real Problem: Taskflow's Nested Subflow Race Condition

The Original Code's Fatal Flaw

The main branch creates a **deeply nested taskflow structure** that looks like this:

```
subset_flow (tf::Taskflow)
     processQueries_task (tf::Subflow& sf)
          for each query:
               query_task (tf::Subflow& query_sf)  <-- NESTED SUBFLOW
                    for each fragment:
                         fragment_task
```

The Race Condition: When you have nested subflows like this: 1. The inner `query_sf.join()` is called to wait for fragments 2. But the outer subflow (`sf`) continues spawning MORE query tasks 3. The executor has multiple subflows in flight simultaneously 4. When `executor.run(*subset_flow).wait()` is called, it may return before all nested work completes

This is why the code has this hack:

```
// Wait for all tasks to complete
std::chrono::milliseconds wait_time(100);
while (executor.num_topologies() > 0) {
    std::this_thread::sleep_for(wait_time);
}
```

But `executor.num_topologies()` doesn't reliably track nested subflows!

Thread-Local FAIDX Reader Destruction Race

The original code uses thread-local FAIDX readers:

```
thread_local ReaderWrapper wrapper;
```

The Problem: - Tasks are creating readers in thread-local storage - When a thread finishes its task and gets recycled by taskflow, the destructor runs - But if file reading is still happening in another part of the subflow, BOOM - the reader is gone - This manifests as truncated output because queries fail to read their sequences

Why Shuo's Solution Works

1. No Nested Subflows = No Race

Shuo's code uses a **flat taskflow**:

```
tf::Taskflow taskflow;  // Simple, flat
for (auto& queryData : allQueryResults) {
    auto task = taskflow.emplace([...]() {
        // Direct work, no nested subflows
    });
}
executor.run(taskflow).wait();  // Single, reliable wait point
```

Why this fixes it: - `executor.run(taskflow).wait()` actually waits for ALL tasks - No nested subflows means no ambiguity about what's complete - The executor can properly track all work

2. Pre-Loading Eliminates Reader Races

By loading all sequences BEFORE parallel processing:

```
// SEQUENTIAL loading phase - single reader, no races
faidx_reader_t* reader = faidx_reader_create(query_meta);
for (const auto& queryName : querySequenceNames) {
    char* seq_data = faidx_reader_fetch_seq(reader, ...);
    queryResults.sequence = std::string(seq_data, seq_len);  // COPY the data
    free(seq_data);
}
faidx_reader_destroy(reader);

// THEN parallel processing - no file I/O needed
executor.run(taskflow).wait();
```

Why this fixes it: - No thread-local readers that can be destroyed prematurely - No concurrent file access - All data is in memory before parallel phase starts - Even if a task fails, the data is still there

3. Semaphore Control Prevents Task Explosion

The original code can create: - N query tasks × M fragment tasks = potentially thousands of tasks

Shuo's code limits concurrency:

```
std::counting_semaphore<8> queryProcessingSemaphore{8};
queryProcessingSemaphore.acquire();  // Max 8 queries in flight
// ... process query ...
queryProcessingSemaphore.release();
```

Why this helps: - Prevents overwhelming the taskflow executor - Reduces memory pressure - Makes task completion more predictable

The Smoking Gun: Why It Fails on Some Systems

The original code's race conditions are **timing-dependent**. They manifest when:

1. **Slow file I/O**: Network filesystems make FAIDX reads slower, increasing the window for thread-local reader destruction
2. **Different thread scheduling**: Some systems recycle threads more aggressively
3. **Memory pressure**: With unbounded tasks, some systems start failing allocations

Shuo's solution removes ALL these timing dependencies by: - Doing I/O sequentially upfront (no race possible) - Using flat taskflow (predictable completion) - Limiting concurrency (bounded resources)

Why Your Work-Stealing Didn't Fix It

Your work-stealing approach still had: 1. **Nested structure** via `processQueryWithFragments` using subflows 2. **Thread-local readers** that could be destroyed 3. **Concurrent file I/O** during the parallel phase

Even though you bounded the workers, the fundamental races remained.

The Counter-Intuitive Truth

Pre-loading everything isn't about memory efficiency - it's about eliminating race conditions.

The "inefficient" approach of loading all data upfront actually makes the code: - **Deterministic**: No timing-dependent behavior - **Reliable**: No reader lifecycle issues
- **Simple**: Flat task graph that taskflow can properly track

Proof This Is The Issue

You can likely reproduce the truncation bug in the original code by: 1. Running on a slow filesystem (network mount) 2. Using many small queries (more task creation overhead) 3. Adding random delays in the FAIDX reader operations

And you'll find Shuo's version is immune to all these conditions because it doesn't do concurrent I/O or use nested subflows.