The Clean Fix: Three Simple Changes

The Problem

1. **Nested subflows** - `executor.run().wait()` can't track nested work properly
2. **Thread-local readers getting destroyed** - When taskflow recycles threads, destructors fire
3. **Task explosion** - Creating tasks for every fragment causes unpredictable completion

The Solution: Three Changes

Change 1: Make Thread-Local Readers Static

Before:

```
thread_local ReaderWrapper wrapper;  // Gets destroyed when task completes
```

After:

```
static thread_local ReaderWrapper wrapper;  // Persists for thread's lifetime
```

Change 2: Flatten the Taskflow

Before:

```
auto subset_flow = std::make_shared<tf::Taskflow>();
auto processQueries_task = subset_flow->emplace([](tf::Subflow& sf) {
    for (const auto& queryName : querySequenceNames) {
        auto query_task = sf.emplace([](tf::Subflow& query_sf) {  // NESTED!
            for (int i = 0; i < fragmentCount; i++) {
                query_sf.emplace([]{...});  // DOUBLE NESTED!
            }
            query_sf.join();
        });
    }
});
executor.run(*subset_flow).wait();
```

After:

```
tf::Taskflow taskflow;
for (const auto& queryName : querySequenceNames) {
    taskflow.emplace([queryName]() {
        // Process entire query in one task
    });
}
executor.run(taskflow).wait();  // Single level, single wait
```

Change 3: Process Fragments Inline

Before:

```
// Inside query task
for (int i = 0; i < fragmentCount; i++) {
    query_sf.emplace([i]() {  // Creating a task per fragment
        processFragment(i);
    });
```

```
}
query_sf.join();
```

After:

```
// Inside query task
for (int i = 0; i < fragmentCount; i++) {
    processFragment(i);  // Just call it directly
}
```

Complete Implementation

```
void mapQuery() {
    tf::Executor executor(param.threads);

    // Static thread-local ensures readers persist
    static thread_local struct {
        faidx_reader_t* reader = nullptr;
        faidx_meta_t* current_meta = nullptr;

        faidx_reader_t* get(faidx_meta_t* meta) {
            if (!reader || current_meta != meta) {
                if (reader) faidx_reader_destroy(reader);
                reader = faidx_reader_create(meta);
                current_meta = meta;
            }
            return reader;
        }

        ~decltype(reader)() {
            if (reader) faidx_reader_destroy(reader);
        }
    } tls_reader;

    // For each subset of references
    for (const auto& target_subset : target_subsets) {
        // Build index for this subset
        buildIndex(target_subset);

        // FLAT taskflow - no subflows
        tf::Taskflow taskflow;

        // Create one task per query (not one task that processes all queries)
        for (const auto& queryName : querySequenceNames) {
            taskflow.emplace([this, queryName, &tls_reader]() {
                // Get thread-local reader
                faidx_reader_t* reader = tls_reader.get(query_meta);

                // Read sequence
                hts_pos_t len;
                char* data = faidx_reader_fetch_seq(reader, queryName.c_str(), 0, -1, &len);
                if (!data) return;

                std::string sequence(data, len);
                free(data);
```

```
                // Process ALL fragments inline
                MappingResultsVector_t results;
                int fragmentCount = sequence.length() / param.windowLength;

                for (int i = 0; i < fragmentCount; i++) {
                    // Direct processing - no task creation
                    const char* fragment = sequence.c_str() + (i * param.windowLength);
                    int fragLen = std::min(param.windowLength, (int)(sequence.length() - i * param.windo

                    // Use existing fragment processing logic
                    QueryMetaData<MinVec_Type> Q;
                    Q.seq = const_cast<char*>(fragment);
                    Q.len = fragLen;
                    // ... rest of fragment processing

                    std::vector<IntervalPoint> intervalPoints;
                    std::vector<L1_candidateLocus_t> l1Mappings;
                    MappingResultsVector_t l2Mappings;

                    mapSingleQueryFrag(Q, intervalPoints, l1Mappings, l2Mappings);

                    // Collect results
                    for (auto& mapping : l2Mappings) {
                        mapping.queryStartPos += i * param.windowLength;
                        results.push_back(mapping);
                    }
                }

                // Handle last fragment if needed
                if (fragmentCount > 0 && sequence.length() % param.windowLength != 0) {
                    // Process final overlapping fragment
                    // ... same inline processing
                }

                // Filter and output results
                auto filtered = filterMappings(results);
                {
                    std::lock_guard<std::mutex> lock(output_mutex);
                    outputMappings(queryName, filtered);
                }
            });
        }

        // Single wait point that actually works
        executor.run(taskflow).wait();

        // Clean up index
        delete refSketch;
    }
}
```

Why This Works

1. **Static thread-local readers** survive across tasks because they're tied to the thread, not the task
2. **Flat taskflow** means `executor.run().wait()` can properly track all work
3. **Inline fragment processing** eliminates task explosion and nested synchronization

What We're NOT Doing

- NOT pre-loading all sequences (memory efficient)
- NOT using mutex on readers (performance preserved)

- NOT creating our own thread pool (using taskflow as intended)
- NOT creating tasks for fragments (predictable task count)

Implementation Steps

1. Remove all `tf::Subflow` parameters and usage
2. Change `thread_local` to `static thread_local` for readers
3. Move fragment processing from task creation to direct function calls
4. Keep the persistent output stream fixes (those were actually good)

This is minimal, clean, and should fix the truncation issue without the memory overhead of pre-loading everything.