Better Solution: Thread-Local Readers Done Right

The Constraint: Thread-Local FAIDX is Required

You're right - FAIDX readers MUST be thread-local for performance. A mutex would serialize all I/O and kill throughput.

The Real Problem We Need to Solve

1. **Nested subflows break completion tracking**
2. **Thread-local readers get destroyed when threads are recycled**
3. **Tasks creating tasks dynamically = unpredictable completion**

Solution: Thread Pool with Persistent Readers

Instead of letting taskflow manage threads (and destroy our readers), we manage our own thread pool:

```
class QueryProcessor {
private:
    struct WorkerThread {
        std::thread thread;
        faidx_reader_t* reader = nullptr;
        std::atomic<bool> should_stop{false};
        std::queue<std::string> work_queue;
        std::mutex queue_mutex;
        std::condition_variable cv;
    };

    std::vector<std::unique_ptr<WorkerThread>> workers;
    std::atomic<size_t> next_query{0};
    std::vector<std::string> queryNames;

public:
    QueryProcessor(int num_threads, faidx_meta_t* meta,
                   const std::vector<std::string>& queries)
        : queryNames(queries) {

        // Create worker threads with PERSISTENT readers
        for (int i = 0; i < num_threads; i++) {
            auto worker = std::make_unique<WorkerThread>();
            worker->reader = faidx_reader_create(meta);  // Each thread gets its own

            worker->thread = std::thread([this, w = worker.get(), meta]() {
                // This thread OWNS this reader for its entire lifetime
                while (!w->should_stop) {
                    // Steal work
                    size_t idx = next_query.fetch_add(1);
                    if (idx >= queryNames.size()) break;

                    const auto& queryName = queryNames[idx];

                    // Read sequence with OUR reader (no destruction risk)
                    hts_pos_t len;
```

```
                    char* data = faidx_reader_fetch_seq(w->reader, queryName.c_str(), 0, -1, &len);
                    std::string sequence(data, len);
                    free(data);

                    // Process fragments inline (no nested tasks)
                    processAllFragments(queryName, sequence);
                }

                // Clean up reader when thread ACTUALLY ends
                faidx_reader_destroy(w->reader);
            });

            workers.push_back(std::move(worker));
        }
    }

    void wait() {
        for (auto& w : workers) {
            w->should_stop = true;
            w->cv.notify_all();
            w->thread.join();
        }
    }
};
```

Alternative: Fix Taskflow's Thread-Local Management

If we want to keep using taskflow, we need to ensure readers aren't destroyed prematurely:

```
// Global registry of readers that outlive taskflow execution
class ReaderRegistry {
private:
    struct ReaderHandle {
        faidx_reader_t* reader;
        std::thread::id owner;
        bool in_use = false;
    };

    std::vector<std::unique_ptr<ReaderHandle>> readers;
    std::mutex mutex;
    faidx_meta_t* meta;

public:
    ReaderRegistry(faidx_meta_t* m, size_t max_threads) : meta(m) {
        // Pre-create readers for all possible threads
        for (size_t i = 0; i < max_threads * 2; i++) {  // 2x for safety
            auto handle = std::make_unique<ReaderHandle>();
            handle->reader = faidx_reader_create(meta);
            readers.push_back(std::move(handle));
        }
    }

    ~ReaderRegistry() {
        for (auto& h : readers) {
```

```
            if (h->reader) faidx_reader_destroy(h->reader);
        }
    }

    faidx_reader_t* getReader() {
        auto tid = std::this_thread::get_id();

        // Try to find existing reader for this thread
        for (auto& h : readers) {
            if (h->owner == tid && h->in_use) {
                return h->reader;
            }
        }

        // Claim an unused reader
        std::lock_guard<std::mutex> lock(mutex);
        for (auto& h : readers) {
            if (!h->in_use) {
                h->owner = tid;
                h->in_use = true;
                return h->reader;
            }
        }

        // Shouldn't happen if we sized correctly
        throw std::runtime_error("Out of readers!");
    }

    void releaseReader() {
        auto tid = std::this_thread::get_id();
        std::lock_guard<std::mutex> lock(mutex);
        for (auto& h : readers) {
            if (h->owner == tid) {
                h->in_use = false;
                break;
            }
        }
    }
};

// Use it with FLAT taskflow
void processWithTaskflow() {
    ReaderRegistry registry(meta, param.threads);
    tf::Taskflow taskflow;

    for (const auto& queryName : querySequenceNames) {
        taskflow.emplace([&registry, queryName]() {
            // Get a reader that WON'T be destroyed
            faidx_reader_t* reader = registry.getReader();

            // Read sequence
            hts_pos_t len;
            char* data = faidx_reader_fetch_seq(reader, queryName.c_str(), 0, -1, &len);
            std::string sequence(data, len);
```

```
            free(data);

            // Process ALL fragments inline (no nested tasks!)
            for (int i = 0; i < fragmentCount; i++) {
                processFragment(sequence, i);
            }

            outputResults(queryName);

            // Don't destroy reader - registry owns it
        });
    }

    executor.run(taskflow).wait();  // Single flat wait
    // Registry destructor cleans up readers
}
```

The Simplest Fix: Static Thread-Local with Flat Taskflow

Actually, the simplest might be to just ensure the thread-local readers persist beyond taskflow execution:

```
// Make readers static so they outlive taskflow runs
static thread_local struct {
    faidx_reader_t* reader = nullptr;
    faidx_meta_t* meta = nullptr;

    void init(faidx_meta_t* m) {
        if (!reader || meta != m) {
            if (reader) faidx_reader_destroy(reader);
            meta = m;
            reader = faidx_reader_create(meta);
        }
    }

    ~ReaderWrapper() {
        // This destructor only runs at thread exit, not task completion
        if (reader) faidx_reader_destroy(reader);
    }
} tls_reader;

void processQueries() {
    tf::Taskflow taskflow;  // FLAT - no subflows!

    for (const auto& queryName : querySequenceNames) {
        taskflow.emplace([meta, queryName]() {
            // Initialize thread-local reader if needed
            tls_reader.init(meta);

            // Read with thread-local reader
            hts_pos_t len;
            char* data = faidx_reader_fetch_seq(tls_reader.reader, queryName.c_str(), 0, -1, &len);
            std::string sequence(data, len);
            free(data);
```

```cpp
            // Process fragments INLINE
            MappingResultsVector_t results;
            for (int i = 0; i < fragmentCount; i++) {
                processFragmentInline(sequence, i, results);
            }

            // Output
            outputResults(queryName, results);
        });
    }

    executor.run(taskflow).wait();  // Works because it's flat!
}
```

The Key Insights

1. **FLAT TASKFLOW IS NON-NEGOTIABLE** - Nested subflows break completion tracking
2. **Thread-local readers are fine IF** they outlive the taskflow execution
3. **Process fragments inline** - Don't create fragment tasks

Recommended Approach

Step 1: Make taskflow flat

Remove ALL `tf::Subflow` usage. Just use a single level of tasks.

Step 2: Fix thread-local reader lifetime

Use `static thread_local` so readers persist across taskflow runs.

Step 3: Process fragments inline

Don't spawn tasks for fragments. Process them in a loop within the query task.

Example Diff:

```diff
- auto processQueries_task = subset_flow->emplace([](tf::Subflow& sf) {
-     thread_local ReaderWrapper wrapper;  // DESTROYED when task ends!
-     for (const auto& queryName : querySequenceNames) {
-         auto query_task = sf.emplace([](tf::Subflow& query_sf) {
-             for (fragment) {
-                 query_sf.emplace([]{...});  // NESTED TASK
-             }
-             query_sf.join();
-         });
-     }
- });

+ static thread_local ReaderWrapper wrapper;  // PERSISTS across tasks
+ tf::Taskflow taskflow;  // FLAT
+ for (const auto& queryName : querySequenceNames) {
+     taskflow.emplace([queryName]() {
```

```
+         wrapper.init(meta);  // Use persistent reader
+         // Read sequence...
+         for (fragment) {
+             processFragmentInline();  // NO TASKS
+         }
+     });
+ }
+ executor.run(taskflow).wait();  // Actually waits!
```

Why This Works

1. **Flat taskflow** = `executor.run().wait()` actually waits for everything
2. **Static thread-local** = Readers don't get destroyed between tasks
3. **Inline fragments** = No task explosion, predictable completion

This gives you: - Thread-local readers (no locking) - Proper completion tracking - No memory explosion from pre-loading everything

Much simpler than Shuo's approach, but solves the actual problems.