Analysis of Mapping Output Truncation Issue in wfmash

Executive Summary

The mapping output truncation issue in wfmash occurs when not all query sequences produce output results, even though the mapping process appears to complete. Three different approaches have been attempted to fix this issue, with varying degrees of success. This document analyzes each approach, identifies the root cause, and proposes an optimal solution.

Problem Description

Issue: Mapping output is truncated - not all query sequences appear in the output PAF file, even though the mapping process seems to run for all queries.

Symptoms: - Missing mappings for some query sequences - Incomplete output when writing to stdout - Potential race conditions in concurrent output writing

Analysis of Three Attempted Fixes

1. fix-stdout-mapping-truncation (Already Merged)

Approach: Fix stream handling for stdout output

Key Changes: - Detect stdout output (`/dev/stdout` or `-`) and use `std::cout` directly - Maintain persistent `ofstream` for file output instead of repeatedly opening/closing - Ensure proper flushing for both stdout and file output - Fix one-to-one filtering to handle stdout properly

Why It Didn't Fully Work: - This fix addressed only the output stream management issue - The root cause may not be just about stdout detection/flushing - Issue persists because the problem is likely in the concurrent task completion/synchronization, not just output handling - While the stream remains open, if tasks aren't completing or their results aren't being properly collected, output will still be truncated

2. high-concurrency-shuo (Shuo's Solution)

Approach: Pre-load all query sequences and optimize concurrency

Key Changes: - **Pre-loads all query sequences into memory upfront** (sequential I/O optimization) - Sorts queries by size (largest first) for better load balancing - Implements semaphore-based stream control (`MAX_CONCURRENT_QUERIES = 8`) - Separates serial vs parallel processing based on query size threshold - Uses thread-local memory pools and pre-allocated buffers - Simplifies taskflow structure - removes nested subflows

Why It Works: - **Sequential I/O**: Loading all queries upfront eliminates concurrent file access issues - **Controlled concurrency**: Semaphore limits memory pressure and prevents task explosion - **Simpler task graph**: Removing complex nested taskflows reduces synchronization issues - **Better resource management**: Pre-allocation and memory pools reduce allocation overhead - **Size-based strategy**: Small queries processed serially avoid overhead of parallelization

Concerns: - **Memory usage**: Loading all queries into memory could be problematic for large datasets - **Not streaming**: Loses ability to process queries as a stream - **Fixed concurrency limit**: Hard-coded semaphore limit may not be optimal for all systems

3. work-stealing-subflows (Your Attempt)

Approach: Implement work-stealing pattern with bounded parallelism

Key Changes: - Creates fixed number of worker tasks (bounded by thread count) - Workers steal queries from atomic counter - Thread-local readers and buffers for efficiency - Processes fragments inline to avoid nested subflow issues - Maintains persistent output streams (incorporated from fix #1)

Why It Didn't Fully Work: - **Fragment processing inline**: Processing fragments sequentially within each query may create bottlenecks - **Thread-local reader lifecycle**: ReaderWrapper with destructor might cause premature cleanup - **Missing synchronization**: May not properly wait for all fragments to complete before outputting - **No query pre-loading**: Still doing concurrent file I/O which can cause issues

Root Cause Analysis

The truncation issue appears to stem from multiple interrelated problems:

1. **Concurrent File I/O**: Multiple threads accessing the FASTA file simultaneously can cause read failures or incomplete data
2. **Task Synchronization**: Complex nested taskflows may not properly wait for all work to complete
3. **Output Stream Management**: Repeated opening/closing or improper flushing of output streams
4. **Memory Pressure**: Task explosion with many queries/fragments can cause resource exhaustion
5. **Race Conditions**: Unsynchronized access to shared resources during result collection

Comparison Matrix

Aspect	fix-stdout	high-concurrency	work-stealing
Stream Management	□ Fixed	□ Fixed	□ Fixed
Query Loading	□ Concurrent	□ Sequential preload	□ Concurrent
Memory Usage	□ Low	□ High (all queries)	□ Low
Concurrency Control	□ Unbounded	□ Semaphore	□ Thread-bounded
Task Complexity	□ Complex nested	□ Simple	□ Moderate
Fragment Parallelism	□ Yes	□ Yes	□ Sequential
Scalability	□ Poor	⚠ Limited by RAM	□ Good

Proposed Optimal Solution

Combine the best aspects of all three approaches:

Core Strategy

1. **Hybrid Loading Strategy**:

```cpp
// Use a bounded queue for queries
class QueryStreamProcessor {
    std::queue<QueryData> queryQueue;
    std::mutex queueMutex;
    std::condition_variable queueCV;
    static constexpr size_t MAX_QUEUE_SIZE = 32; // Configurable

    // Producer thread loads queries
    void loadQueries() {
        faidx_reader_t* reader = ...;
        for (const auto& queryName : querySequenceNames) {
            // Load query
            QueryData data = loadQuery(reader, queryName);

            // Add to queue (blocks if full)
            {
                std::unique_lock lock(queueMutex);
                queueCV.wait(lock, [this] {
                    return queryQueue.size() < MAX_QUEUE_SIZE;
                });
                queryQueue.push(std::move(data));
            }
            queueCV.notify_all();
        }
    }
};
```

2. **Adaptive Processing Strategy**:

   - Use query size and system resources to decide processing strategy
   - Small queries: Process serially (avoid overhead)
   - Medium queries: Process with limited parallelism
   - Large queries: Full parallel processing with fragments

3. **Smart Resource Management**:

```cpp
// Dynamic semaphore based on system resources
size_t getOptimalConcurrency() {
    size_t available_memory = getAvailableMemory();
    size_t query_memory_estimate = averageQuerySize();
    size_t thread_count = param.threads;

    return std::min({
        thread_count * 2,  // Allow some oversubscription
        available_memory / query_memory_estimate,
        size_t(16)  // Reasonable upper bound
    });
}
```

4. **Robust Output Handling**:

   - Use persistent output stream (from fix #1)
   - Implement write queue with dedicated writer thread
   - Ensure ordered output for deterministic results

Implementation Plan

1. **Phase 1: Stream Processing Infrastructure**
    - Implement bounded query queue
    - Add producer thread for query loading
    - Create consumer pool with work-stealing
2. **Phase 2: Adaptive Strategy**
    - Add query size classification
    - Implement strategy selector
    - Create fragment batch processor
3. **Phase 3: Resource Management**
    - Add memory monitoring
    - Implement dynamic concurrency adjustment
    - Add backpressure mechanisms
4. **Phase 4: Output Pipeline**
    - Create output queue
    - Implement writer thread
    - Add result ordering if needed

Recommended Immediate Action

Given the constraints and the fact that Shuo's solution works:

1. **Accept high-concurrency-shuo as the base** - it's proven to work
2. **Optimize memory usage**:
    - Add streaming capability for very large datasets
    - Implement query batching (process N queries at a time)
    - Add memory limit parameter
3. **Make concurrency configurable**:
    - Replace hard-coded semaphore limit with parameter
    - Auto-tune based on system resources
4. **Add fallback mechanism**:
    - Detect memory pressure and switch to streaming mode
    - Provide option to disable pre-loading

Code Example: Enhanced Version

```
class EnhancedMapper {
private:
    struct Config {
        size_t max_queries_in_memory = 100;
        size_t max_concurrent_queries = 8;
        size_t small_query_threshold = 100;
        bool enable_preloading = true;
        bool auto_tune = true;
    };

    Config config;

public:
    void processQueries() {
        if (shouldUsePreloading()) {
            processWithPreloading();  // Shuo's approach
        } else {
            processWithStreaming();   // Memory-efficient approach
        }
    }

private:
    bool shouldUsePreloading() {
        size_t total_query_size = estimateTotalQuerySize();
        size_t available_memory = getAvailableMemory();

        return config.enable_preloading &&
               (total_query_size < available_memory * 0.5);
    }

    void processWithPreloading() {
        // Shuo's implementation with enhancements
        auto queries = loadAllQueries();

        // Auto-tune concurrency
        if (config.auto_tune) {
            config.max_concurrent_queries =
                calculateOptimalConcurrency(queries.size());
        }

        // Process with semaphore control
        processQueriesOptimized(queries);
    }

    void processWithStreaming() {
        // Memory-efficient streaming implementation
        QueryStreamProcessor processor(config);
        processor.run();
    }
};
```

Conclusion

The mapping truncation issue is a complex problem involving concurrent I/O, task synchronization, and resource management. While the high-concurrency-shuo solution works by pre-loading all queries, it trades memory for correctness. The optimal solution would combine:

1. Shuo's proven approach for correctness
2. Streaming capability for memory efficiency
3. Adaptive strategies based on dataset characteristics
4. Configurable resource limits

The immediate path forward is to enhance Shuo's solution with better resource management while maintaining its correctness guarantees.