Parallel Text Reference Breaker

Efficient Wikipedia Reference Detection using Aho-Corasick Algorithm and OpenMP

Rui Cai

Parallel Programming Course Project 2025

May 30, 2025

Outline

Project Overview

- **Goal**: Automatically identify and annotate Wikipedia references in large text files
- **Approach**: Parallel text processing using Aho-Corasick string matching algorithm
- **Implementation**: C++11 with OpenMP parallelization
- **Output**: JSON format with reference annotations and positions

Key Features

- Offline Wikipedia titles database matching
- Parallel processing with configurable thread count
- Memory-efficient immutable trie structure
- 1KB block-based text segmentation

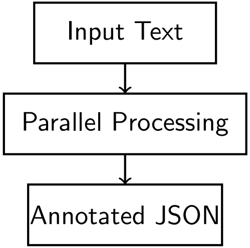
Problem Statement

Challenge:

- Large text files need semantic annotation
- Manual reference identification is time-consuming
- Need to match against millions of Wikipedia titles
- Performance requirements for real-time processing

Solution:

- Automated reference detection
- Parallel processing for scalability
- Efficient string matching algorithm
- Structured JSON output format



Usage Example

Command Line Interface

```
# Basic usage
./text_reference_breaker wiki_titles.txt input.txt output.json

# Sequential mode (for comparison)
./text_reference_breaker wiki_titles.txt input.txt output.json --sequential

# Example with timing
time ./text_reference_breaker data/wiki_titles.txt large_text.txt result.json
```

JSON Output Format

```
{
  "text": "Coldplay␣had␣a␣tour␣concert␣in␣Helsinki␣in␣August,␣2024.",
  "references": [
    { "start": 0, "end": 8, "titleIndex": 301, "title": "Coldplay" },
    { "start": 28, "end": 36, "titleIndex": 2105, "title": "Helsinki" },
    { "start": 40, "end": 46, "titleIndex": 984, "title": "August" }
  ]
```

Applications and Use Cases

Academic and Research

- Automatic citation generation for research papers
- Content analysis and topic modeling
- Knowledge graph construction

Content Management

- Wiki-style reference systems
- Educational content annotation
- News article fact-checking preparation

Data Processing

- Large-scale text corpus analysis
- Information extraction pipelines
- Semantic text enrichment

Aho-Corasick Algorithm

Time Complexity Analysis:

Construction Phase (Sequential)

- Build trie and failure links: $O(m)$
- Where $m = \sum_{i=0}^{|P|} |title_i|$ (total pattern length)
- Run once, immutable afterward

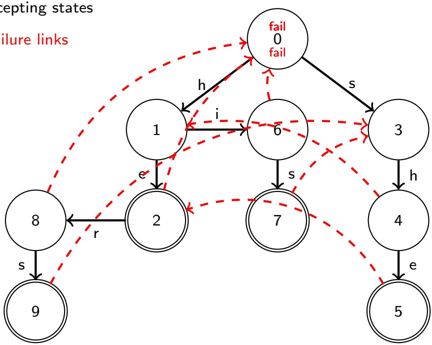
Search Phase (Parallelizable)

- Text traversal: $O(n+k)$
- Where n = text length, k = number of matches
- Each thread processes independent chunks

Patterns: "he", "she", "his", "hers"

Double circles = accepting states

Red dashed = failure links



Trie with failure links

Data Structures

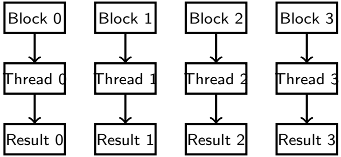
Core Data Structures

```
struct Reference {
    size_t start;           // Start position in text
    size_t end;             // End position in text
    size_t titleIndex;      // Index in Wikipedia titles
    std::string title;      // Matched Wikipedia title
};
struct ProcessedBlock {
    std::string text;                    // Block content
    std::vector<Reference> references;   // Found references
};
class TextProcessor {
private:
    std::unique_ptr<aho_corasick::trie> trie_;
    std::unordered_map<std::string, size_t> titleIndices_;
```

Parallelization Strategy

Block-Based Parallelization

1. **Text Segmentation**: Split input into 1KB blocks
2. **Parallel Processing**: Each thread processes independent blocks
3. **Result Aggregation**: Combine results maintaining order
4. **Position Adjustment**: Correct reference positions for global text



Block 0 → Thread 0 → Result 0

Block 1 → Thread 1 → Result 1

Block 2 → Thread 2 → Result 2

Block 3 → Thread 3 → Result 3

OpenMP Implementation

Listing 1: Parallel Processing Code

```
nlohmann::json TextProcessor::processFileParallel(const std::string& inputFile) {
    std::string text = utils::readFile(inputFile);
    std::vector<std::string> blocks = splitIntoBlocks(text);
    std::vector<ProcessedBlock> processedBlocks(blocks.size());

    size_t currentOffset = 0;

    #pragma omp parallel for schedule(dynamic)
    for (size_t i = 0; i < blocks.size(); ++i) {
        size_t blockOffset = 0;
        for (size_t j = 0; j < i; ++j) {
            blockOffset += blocks[j].length();
        }
        processedBlocks[i] = processBlock(blocks[i], blockOffset);
    }

    return blocksToJson(processedBlocks);
}
```

Key OpenMP Features:

- `#pragma omp parallel for` for loop parallelization
- `schedule(dynamic)` for load balancing
- Thread-safe trie operations (read-only after construction)

Performance Considerations

Memory Efficiency:

- Immutable trie structure
- Shared read-only data
- Minimal memory allocation per thread
- Block-based processing reduces memory footprint

Load Balancing:

- Dynamic scheduling
- Variable block processing times
- Automatic work distribution

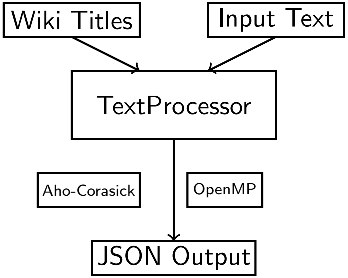
Scalability Factors:

- Thread count = `omp_get_max_threads()`
- No thread synchronization during processing
- Linear speedup potential
- I/O bound for very large files

Cache Efficiency:

- Sequential block processing
- Locality of reference
- Trie fits in cache for common patterns

System Architecture



Key Components:

- **TextProcessor**: Main processing class
- **Aho-Corasick Trie**: Pattern matching engine
- **OpenMP**: Parallelization framework
- **JSON Output**: Structured result format

Build System and Dependencies

CMake Configuration

```
cmake_minimum_required(VERSION 3.10)
project(text_reference_breaker VERSION 1.0)

set(CMAKE_CXX_STANDARD 11)
find_package(OpenMP REQUIRED)

Dependencies:
- nlohmann/json (JSON processing)
- aho_corasick (Header-only string matching)
- OpenMP (Parallelization)
```

External Dependencies:

- **Aho-Corasick**: `github.com/cjgdev/aho_corasick`
- **nlohmann/json**: `github.com/nlohmann/json`
- **Test Data**: `github.com/mmcky/nyu-econ-370`

Performance Benchmarks

Test Environment

- **Hardware**: 16-core system with OpenMP
- **Compiler**: GCC with C++11 and OpenMP support
- **Wikipedia Databases**: 51 titles vs 10,000 titles
- **Input Texts**: 2.2KB sample vs 3.1MB War and Peace
- **Note**: Tables show processing time only (loading time: 6-13ms for 51 titles, 66-79ms for 10K titles)

Small Database (51 titles):

Input	Mode	Processing Time	Speedup
2.2KB	Sequential	5.64ms	-
2.2KB	Parallel	13.95ms	0.40x
3.1MB	Sequential	7863.76ms	-
3.1MB	Parallel	1296.74ms	**6.06x**

Large Database (10K titles):

Input	Mode	Processing Time	Speedup
2.2KB	Sequential	1252.71ms	-
2.2KB	Parallel	671.75ms	**1.76x**
3.1MB	Sequential	1766.43s	-
3.1MB	Parallel	285.80s	**6.18x**

Performance Analysis

Key Findings:

- **Small files**: Sequential faster (overhead dominates)
- **Large files**: Parallel provides 6x speedup consistently
- **Loading overhead**: 10K titles take 12x longer to load than 51 titles (66-79ms vs 6-13ms)
- **Processing complexity**: Large databases require 200x more processing time due to more matches
- **Extreme scaling**: 1766s → 286s for largest test (6.18x speedup)

Scalability Analysis:

- **Consistent speedup**: 6x for large texts regardless of database size
- **Amdahl's Law**: Parallel portion dominates for large texts
- **Loading vs Processing**: Loading is one-time cost, processing scales with text size
- **Memory bandwidth**: Limits not reached even with 5M+ references
- **I/O impact**: Minimal compared to processing time

Why Not Use Full Wikipedia Title Dump?

Scale Challenge

- **Full Wikipedia dump**: 100MB+ title file
- **Current testing**: 51 titles (445 bytes) vs 10,000 titles (3.2MB)
- **Estimated full dump**: 300,000-350,000 titles (30-35x larger than 10K)
- **Time complexity**: Construction $O(m)$, Search $O(n + k)$

Why Not Use Full Wikipedia Title Dump?

Extrapolated Performance:

- **Loading time estimate**:
  - 10K titles: 70ms
  - 300K titles: 2-3 seconds
- **Processing time estimate**:
  - War & Peace (3.1MB text)
  - Sequential: 12-15 hours
  - Parallel: 2-2.5 hours

Practical Considerations:

- **Memory usage**: 100MB+ trie structure
- **Reference density**: 100M+ matches expected
- **Diminishing returns**: Over-annotation problem
- **Development focus**: Algorithm optimization vs data scale

Future Enhancements

Performance Improvements:

- ~~GPU acceleration (CUDA/OpenCL)~~
- SIMD vectorization
- Memory-mapped I/O
- Distributed processing

Feature Extensions:

- **Fuzzy matching support**
- Multiple language support
- Real-time streaming processing
- **Web service API**

Algorithm Enhancements:

- Context-aware matching
- Machine learning integration
- Semantic similarity scoring
- Disambiguation algorithms

Scalability:

- Cloud deployment
- Microservice architecture
- Database integration
- Caching mechanisms

Acknowledgments

Open Source Libraries

- **Aho-Corasick Algorithm**: Header-only C++ implementation by cjgdev
  https://github.com/cjgdev/aho_corasick
- **JSON Processing**: nlohmann/json library by Niels Lohmann
  https://github.com/nlohmann/json
- **OpenMP**: Parallel programming API for shared memory multiprocessing

Data Sources

- **War and Peace Text**: Downloaded from NYU Economics 370 course materials
  Maintained by mmcky at github.com/mmcky/nyu-econ-370
- **Google 10K English Words**: Common English vocabulary from first20hours/google-10000-english
  Based on n-gram frequency analysis of Google's Trillion Word Corpus
- **Wikipedia Titles**: Custom curated sample of 51 popular articles

Questions and Discussion

Thank You!

Questions?

Project Repository: `github.com/RayChromium/parallel_text_breaker`
Technologies Used: C++11, OpenMP, Aho-Corasick, CMake
Key Metrics: Parallel processing, O(n+m+z) complexity, 1KB block size