15-618 Project Final Poster: Parallel String Matching Algorithms

Abhishek Kumar and Runze Wang

Summary

Project objectives

- Implement exact string matching algorithms (KMP, Aho-Corasick, and Rabin-Karp)
- Focus on matching multiple patterns
- Parallelize the algorithms using CUDA and OpenMP frameworks
- Parallelize construction of Aho-Corasick automaton via lock-free data structures and parallel BFS
- Conduct experiments on constructed data to benchmark runtime, cache misses, and branch misses
- Reason about factors limiting the speedup
- Conduct experiments on corpus from bioinformatics: matching against DNA and protein corpuses

Project Motivation

- Text Mining
- Network Intrusion Detection Systems
- Bio-informatics
- Digital forensics
- Plagiarism Detection

All of these applications use String Matching
It’s usually the performance bottleneck

Background

KMP

- Linear time single pattern search algorithm
- Preprocesses pattern to create a Longest Prefix Suffix (LPS) array, also known as “Failure Function”
- Time Complexity *O(n+m)* (n: size of text, m: size of pattern)

Rabin Karp

- Hashing based string search algorithm
- Expected linear time complexity
- Performance can be poor due lot of arithmetic (multiplications)

Aho-Corasick

- Generalization of KMP to multiple patterns
- Finite State Automaton based on Trie
- Additional links to help faster transition between states in case of failures
- Additional fields called “output function”, which maintain the list of patterns that are matched at the given nodes
- Time complexity *O(n+m+z)* (m: sum of sizes of patterns, z: number of occurrences)

Approaches

OpenMP Parallelism

- Simplest parallel programming model
- For KMP and Rabin-Karp, we can simply parallelize over the patterns
- For Aho-Corasick,
    - Insert pattern strings to Trie via a lock-free variant of Trie (based on compare_and_swap primitives)
    - Construct the failure links using parallel BFS traversal
- For all algorithms, we can parallelize the matching phase by dividing the texts into blocks, pad the blocks to allow patterns that spread across the blocks

CUDA Parallelism

- KMP and Rabin-Karp: Iterate over the patterns in parallel, threads in a warp work with the same text chunk to improve locality
- Aho-Corasick: Each thread being responsible for one starting position of the text, designed for data parallelism

Experiment Setup

- Construct test cases to simulate adversarial applications and to showcase weaknesses of different approaches
- Measured cache miss rates and branch miss rates to reason about deviations from theoretical analysis
- Tested implementations on real-word corpus: protein and DNA patterns
- Experiments conducted on GHC machines

Benchmark on Generated Data

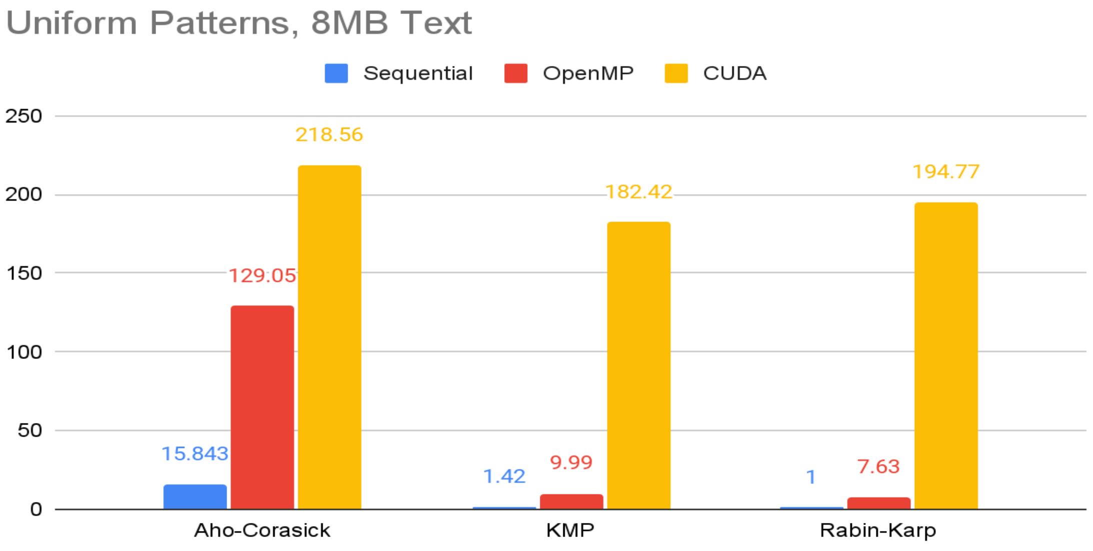
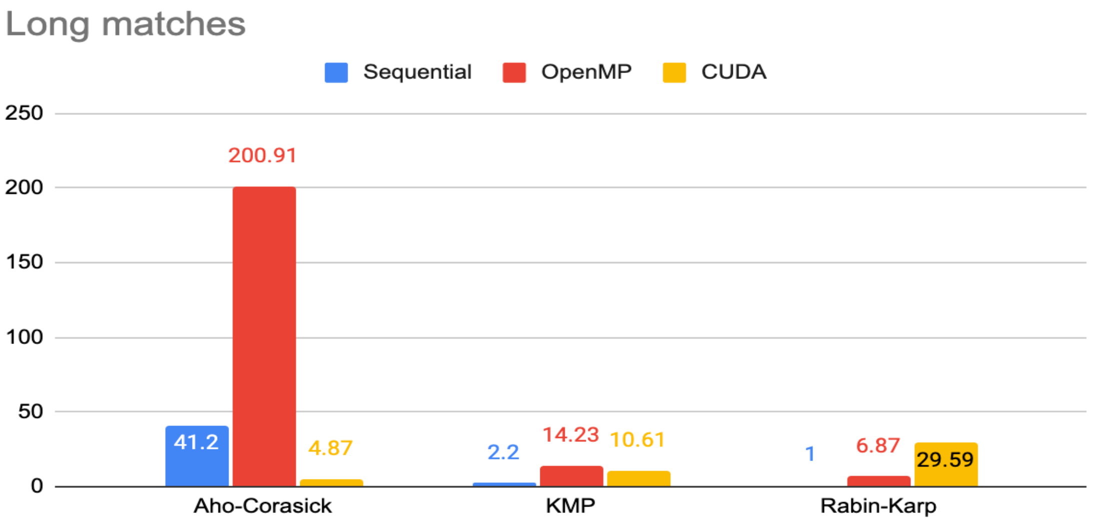


Table 1 Cache Miss statistics
(8MB random text, 24 random patterns of length 15)

Algorithm (Sequential)	Cache References	Cache Misses	Miss Rate
AC	951,657	292,065	30.690%
KMP	949,953	277,047	29.164%
Ratio	0.998	0.948	0.950



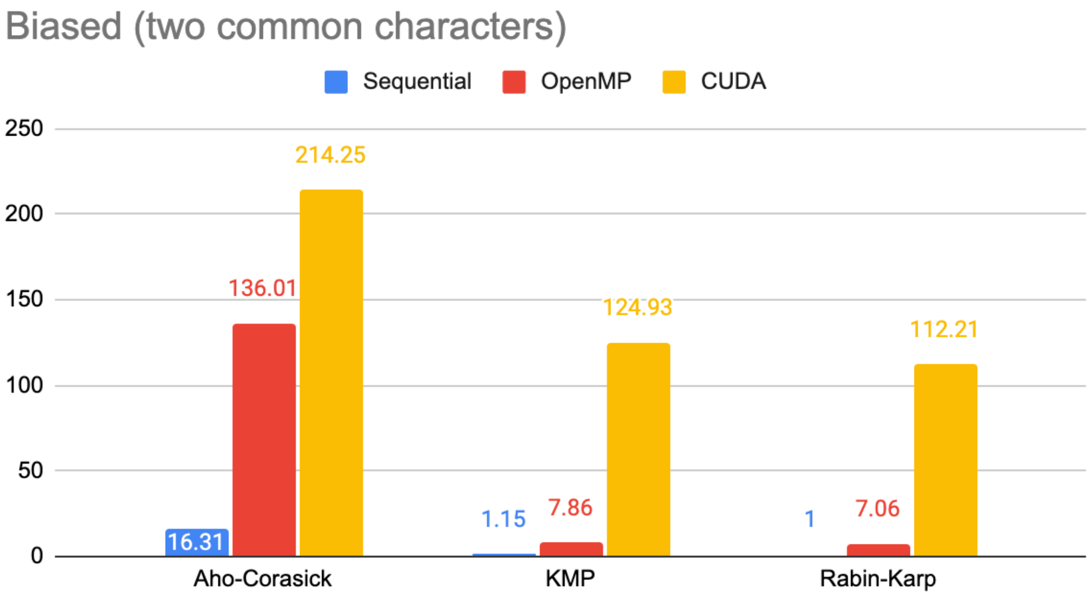
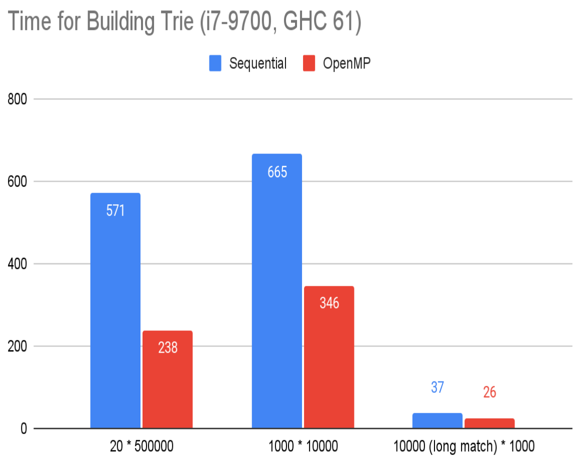
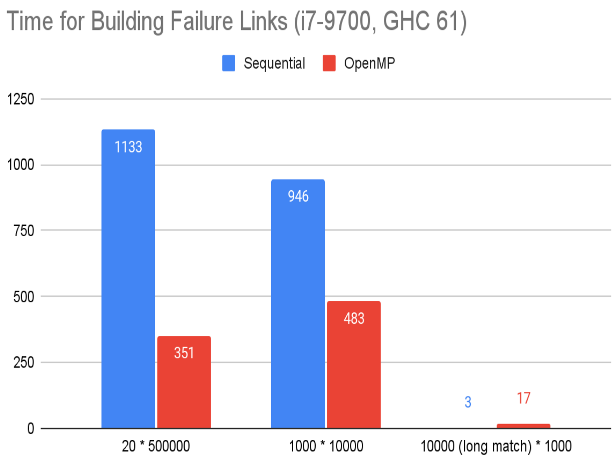


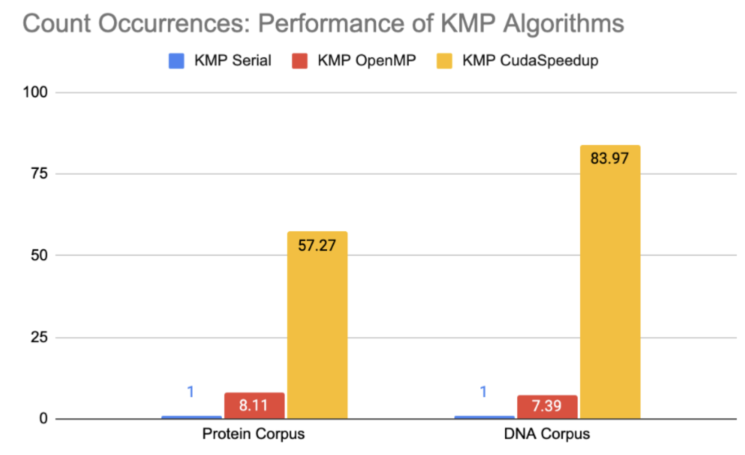
Table 2 Branch Miss statistics
‘a’ and ‘b’ appear with 99% probability, rests are ‘c’ and ‘d’
Force patterns to start with given characters.

Setup	Branches	Branch Misses	Miss Rate
Start with ‘a’, KMP	112,642,009	1,325,748,468	8.50%
Start with ‘c’, KMP	13,333,300	1,445,196,497	0.92%

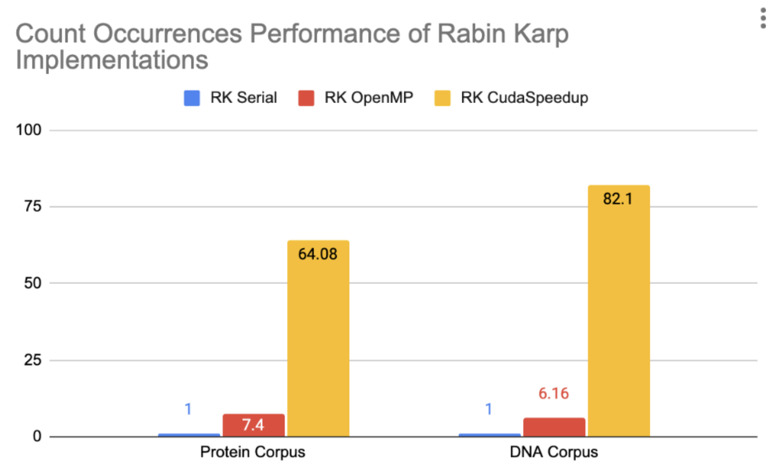




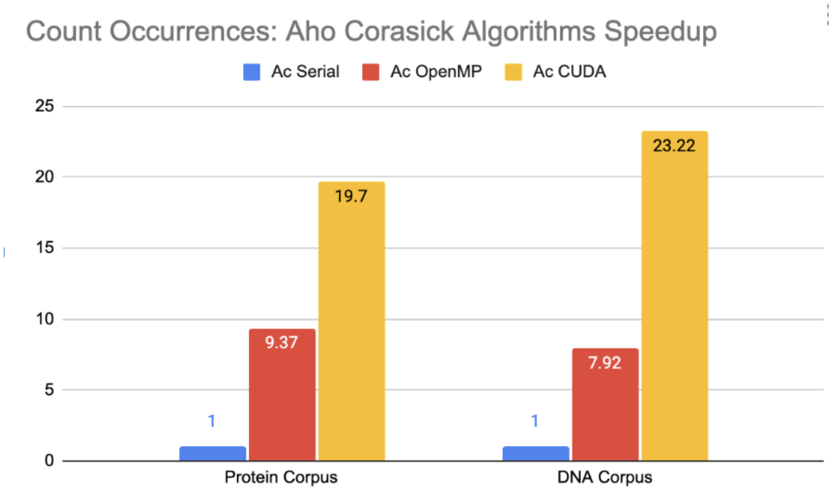
Benchmark on Corpuses : Protein and DNA



Count Occurrences: Performance of KMP Algorithms

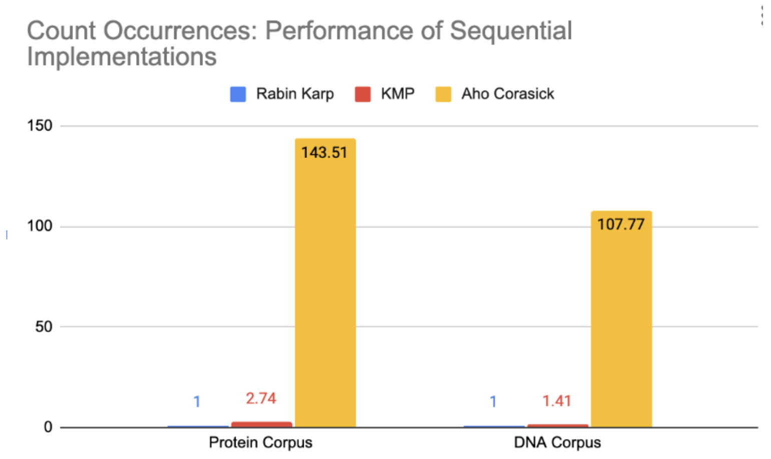


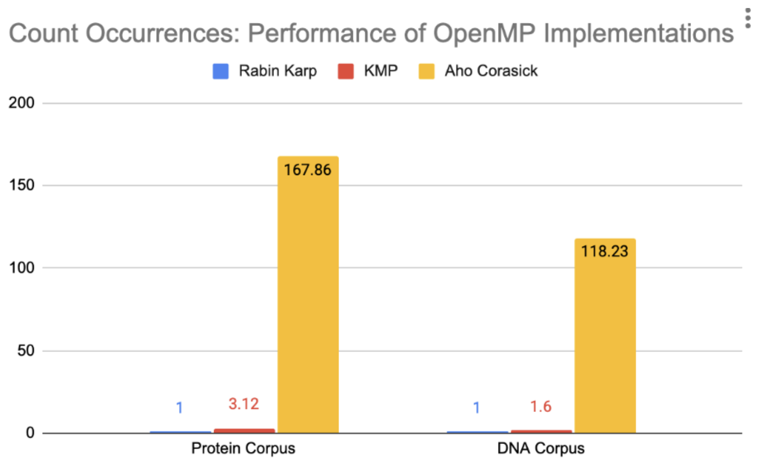
Count Occurrences Performance of Rabin Karp Implementations

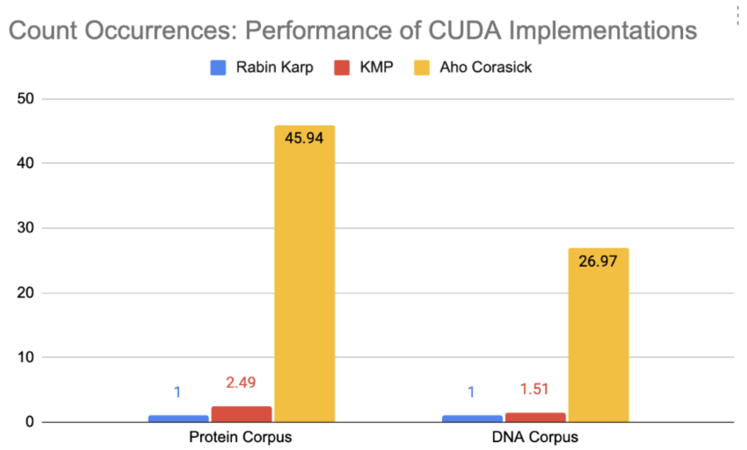


Count Occurrences: Aho Corasick Algorithms Speedup

Benchmark on Corpuses : Protein and DNA


Count Occurrences: Performance of Sequential Implementations


Count Occurrences: Performance of OpenMP Implementations


Count Occurrences: Performance of CUDA Implementations

Conclusion

Sequential/OpenMP algorithms over single pattern

- Rabin-Karp: Performance stable though poor; regular memory access patterns and mostly uniform amount of work for each position of the text; costly arithmetic operations limit performance
- KMP: In general the fastest, though the speedup depends on whether branch predictors lead to huge amount of branch misses
- Aho-Corasick: Slower than KMP due to less compact data structures and higher cache miss rates

Sequential/OpenMP algorithms over multiple pattern

- Aho-Corasick: The obvious choice when there are multiple patterns
- KMP and Rabin-Karp: Slow due to need to loop over the text multiple times

CUDA implementation of Aho-Corasick

- In general the fastest, though the implementation focusing on data parallelism does not seem to be good if patterns are long and for most positions, we can match the patterns for an extended length

CUDA implementations of KMP and Rabin-Karp

- Huge speedup against sequential and OpenMP implementations of KMP and Rabin-Karp, though still not as efficient as Aho-Corasick on most test cases