Parallel String Matching

Dengyu Liang
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
dengyuliang@cmail.carleton.ca

December 25, 2022

Abstract

This project explores and implements CPU parallel using Cilk and GPU parallel using Cuda for seven different string-matching algorithms and evaluates the performance of each algorithm. Each algorithm was evaluated on three different sets of data: natural language text, genomic data, and program-generated data. The results showed that most algorithms can benefit from CPU parallelism, achieving 3-10x speedup in 16 threads. GPU performs poorly on most files less than 100MB and only has a huge speedup on special data, such as the 1000 repeat letters on the brute force algorithms, but still see the speedup in most cases. Based on the CPU and GPU in the experiment, GPU has an 8x speedup on the Boyer-Moore algorithms and 90X speedups in the 1000 repeat letters on the brute force algorithms.

1 Introduction

String matching is a fundamental problem in computer science, with heavy applications in fields such as bioinformatics, search engines, and large database searches. While sequence algorithms are effective at solving this problem, their performance can be limited by the linear increase in the size of the matching string, leading to long search times for large datasets. In recent years, there has been a trend toward using modern CPU with multiple cores and GPU for data processing, and parallelization has become an important consideration in algorithm design. Given the data structure of string matching, it can potentially be highly amenable to parallel programming, allowing for efficient solutions to large-scale matching problems on large-scale systems.

The goal of this project was to investigate the parallel implementation of traditional sequence string matching algorithms, implementing their parallel versions using Cilk and Cuda, and comparing the benefits of different string matching algorithms in different parallel architectures. I also examined the potential applications and limitations of these algorithms on a distributed architecture. In this project, I implemented seven different string-matching algorithms in CPU parallel and five in GPU. For algorithms that are difficult to parallelize algorithmically, I partitioned the index to perform parallel operations. Some of the algorithms also used bit parallel or SSE instructions to improve performance. I analyzed the speedup

obtained by different algorithms and the worst-case speedup obtained.

It can choose a more effective algorithm through analysis of pattern and text length, as well as preprocessing. Our experimental results also demonstrate this point. The type of data being used can also impact efficiency, as a genomic sequence with four letters may have more repetitions than natural language text, which can affect the performance of the algorithm. I will compare different datasets to explore the differences and determine the best algorithm.

2 Literature Review

String Matching is an important problem in computer science, and many people studied extensively. The most recent comparison was Philip Pfaffe, who discussed 7 parallelized string matching algorithms based on SIMD[1]. The Flynn classification divides computer architectures into four categories[17], Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data, and for different architectures, there have different methods to minimize latency and improve performance. Moreover, the CPU and GPU architectures also have different memory hierarchies, which need to be taken into account when designing string-matching algorithms. This project compares the advantages and disadvantages of different architectures in solving string-matching problems and measures the performance of CPU parallelism and GPU parallelism.

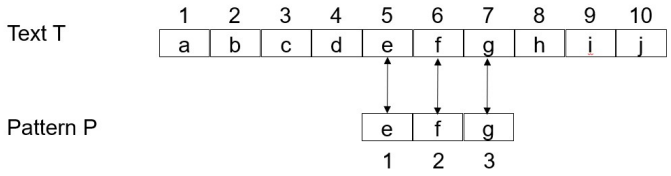

Figure 1: String Matching wants to find the pattern from Text

2.1 Sequential String Matching

String Matching is one of the common fundamental problems, and many algorithms have been proposed to solve it. Faster solutions were proposed as early as 1977, the KMP algorithm and the Boyer–Moore algorithm[1]. Since 1970, more than 80 String Matching algorithms have been proposed. These algorithms use preprocessing techniques to speed up the search process. In addition, algorithms such as the edit distance algorithm and suffix tree can be used to solve more complex string-matching problems. Based on previous research those algorithm deformations, combinations, and expansion, String Matching has been continuously optimized. Now, using parallelized algorithms to improve performance is a more mainstream

method[1], by running the traditional algorithm in parallel, the original algorithm can be accelerated. The specific algorithm is listed in section 4 Implementation. By exploring the details of these algorithms, the efficiency of parallelized algorithms can be further improved.

2.2 Parallel String Matching

Parallel computing has great potential and extremely high scalability. Making good use of parallel computing can greatly speed up the calculation speed. Although not all algorithms can benefit from parallel computing. Specific to the String Matching algorithm, It can be parallelized from data input, It is also possible to parallelize certain operations. In conclusion, parallel computing has great potential and can be used to speed up the calculation speed of string-matching algorithms. However, depending on the algorithm, the best core utilization can vary and it is important to analyze the speedup achieved in different numbers of threads.

2.2.1 Parallel String Matching base on MISD

Although there exist MISD architectures dedicated to pattern matching, such as Halaas's recursive architecture [6], which can be used to solve string-matching problems, its performance is not competitive compared to other solutions. It can match up to 127 patterns at a clock frequency of 100 MHz, but this is not much better than a 1GHZ single-core CPU. Moreover, MISD processors lack practical applications, and research in this area has been shelved.

2.2.2 Parallel String Matching base on SIMD

SIMD uses one instruction stream to process multiple data streams, which can lead to significant performance improvements compared to executing the same instruction on each data element sequentially. Most Parallel String Matching benefits from SIMD architecture. Such as Chinta Someswararao's Butterfly Model [9], and some string matching algorithms are developed based on the SSE instruction set [1], for example, the SSEF algorithm. which is a SIMD instruction set developed by Intel. Another typical SIMD device is the GPU.

2.2.3 Parallel String Matching base on GPU(CUDA)

Compared with the CPU, modern GPU has huge advantages in parallel computing. A single GPU integrates thousands of computing units, so thousands of parallel computing can be realized on a single GPU. There are also some ways to use CUDA programming provided by Nvidia GPU for String Matching, Giorgos Vasiliadis implements string search and regular expression matching operations for real-time inspection of network packets through the pattern matching library [2], And in 2011 it reached a data volume close to 30 Gbit/s. Efficient GPU algorithms can be 30 times faster than a single CPU core, and Approximate String Matching with k Mismatches reports an 80 times speedup [3]. This makes GPU computing have certain advantages in cost and speed, but the research in this area is not as much as the traditional string algorithm.

2.2.4 Parallel String Matching base on MIMD

MIMD machines can execute multiple instruction streams at the same time. Many modern CPUs belong to this type. In order to fully mobilize multiple instruction streams, targeted

parallel algorithms are inevitable. There are not many studies in this area. Hitoshi developed an algorithm called PZLAST used for MIMD processor PEZY-SC2 and compared the performance of BLAST+, CLAST, and PZLAST algorithms[11], which are specially optimized for the biological field. However, they didn't explore the percentage of parallel utilization of the algorithm.

2.2.5 Distributed Memory Parallel String Matching

There are few articles discussing the String Matching algorithm on Shared and Distributed-Memory Parallel Architectures, except Antonino Tumeo's Aho-Corasick algorithm in 2012 compared and analyzed the distributed memory architecture and shared memory architecture[10]. Results at the time showed that shared-memory architectures based on multiprocessors were the theoretical best performance, but there is an upper limit to the amount of parallelism, at 80 threads he starts to get degraded speedup, and beyond 96 threads the speedup becomes marginal. considering cost constraints, GPUs that did not reach the PCI-Express bandwidth limit had the best price/performance ratio. The performance of GPU has developed several times in the past ten years, and theoretically, its performance is far faster than that of CPU now. Although distributed can provide sufficient space and computing resources, limited by the cost of communication, the performance of distributed computing is not satisfactory. Especially for a single String Matching problem. Nonetheless, this can lead to considerable performance gains for multiple patterns search, as shown by Panagiotis' work[8], In a 16-processor cluster, the matching time can be reduced by half of the original.

3 Problem Statement

A definition of the problem of string matching is to find a pattern P in a text T. I define the pattern as having length p and the text as having length t, with both the pattern and text based on the same alphabet. The output is the position of every occurrence of P in T. The number of threads used is c. The input is dynamic, and the time cost of preprocessing the pattern or text should be included in the runtime. I are only concerned with finding exact matches like Figure 1, and are not considering the problem of finding approximate matches or regular expression patterns.

4 Implementation

The algorithms explore range from the most basic brute force algorithms, and partial optimization algorithms to modern advanced algorithms using SSE and bit parallelism. In the parallel implementation of the algorithm, except for the Brute Force algorithm, which is a direct parallel loop, the CPU of other algorithms uses the method of index segmentation in parallel, so that each thread is allocated to a string of(t/c + p - 1), thereby realizing parallel operations. The slice algorithm is shown in Algorithm 1. The parallel implementation of the GPU is to limit the index range so that one thread accesses n string lengths. In this case, the preprocessing of the algorithms needs to be done ahead of time on the CPU, while modifying all algorithms so that they are bounded by scope per thread. Some of the algorithms use the SSE instruction set to improve bit parallelism, Due to the use of SSE instructions, SSEF and

EPSM are not parallelizable on the GPU. The following subsections provide a brief overview of the implemented algorithms.

Algorithm 1: Slice text by thread

Data: Pattern, Text, Total Threads
Result: count of the matching number
$i \leftarrow 0$;
while $i \leq (tread - 1)$ **do**
  $T \leftarrow Text[i \times \frac{N}{tread}, i \times \frac{N+1}{tread} + N - 1]$
  i++;
  Algorithm(Pattern, T); ; `/* parallel algorithm */`
end

4.1 Brute Force

The basic Brute Force algorithm compares the pattern to the text, one character at a time until unmatching characters are found. Brute Force means the worst algorithm in most problems. But that's not exactly the case in String Matching Problem. In the average case, Brute Force takes O (p + t) steps. The result reflected in the experiment is that in some natural language searches, the Brute Force algorithm is faster than some algorithms. But there's no way to avoid bad worst-case outcomes. In the worst case, the Brute Force algorithm requires (p*t) steps, which is unacceptable in large searches. Many algorithms benefit from the pattern to avoid the worst case but brute force algorithms do not.

4.2 KMP algorithm

KMP algorithm, maybe the first optimization algorithm[18] for string match algorithm, KMP calculates the number of times it can skip each time it reads a string, It uses a preprocessing phase on the pattern to build a partial match table. Although this avoids the worst result, In theory, KMP has a time cost of O(t), but as an early version of the optimization algorithm, KMP can't speed up too much. Excellent performance in some cases, but doesn't take high scores in most cases. relatively, due to more calculations, it gets more benefit from parallelism.

4.3 Boyer-Moore

Boyer-Moore algorithm calculates the bad character rule and the good suffix rule[14], matches from the last index to the front, and skips unmatch string by rule. This algorithm takes advantage of the fact that most strings don't match the pattern, and skips large portions of the string without checking. Thus exceed linear time in algorithm analysis for most natural languages and large pattern capacity matching problems with better speedup than KMP. it has (p) Preprocessing time and (t/p) at best, O(p*t) at the worst Matching time, and requires (p) space to store the good and bad rules.

4.4 Rabin-Karp

The Rabin-Karp algorithm gets inspiration from the hash function. When scanning strings for comparison, it uses the hash value generated by Rolling hash comparison instead of directly comparing strings[13]. This avoids directly scanning the entire pattern. Only strings with the same hash can be compared. A full scan will be performed. Because the hash is fixed, the speed of the algorithm depends on the hash function and the string alphabet. The worst result is p*t(p depending on hash function), but as a heuristic algorithm, it is faster than the expected value of the brute force algorithm.

4.5 Shift-Or

The Shift Or algorithm uses bitwise techniques[1], it keeps an array of bits, R, showing if prefixes of the pattern don't match at the current place. Before searching, mismatch arrays are computed for each character in the alphabet and saved in an array, S. For the next position, with the character c, R = shift(R) or S[c]. If the last bit of R is 0, the pattern matches. The runtime is deterministic and in O(n*m). Nevertheless, it is a fast operation for short pattern due to bit parallelism.

4.6 SSEF

The SSEF algorithm precomputes 65536 filter lists based on the kth bit of each character on the patterns[1]. These filters are then applied efficiently, utilizing SSE instructions, on shifting alignments of pattern and text. SSEF is restricted to patterns with a minimum length of m greater than 32. The worst-case runtime is in O(n*m). Considering the probability to filter possible matches, SSEF achieves an average runtime in O(n*m/65536). The SSEF algorithm doesn't exactly cover all implementations, but it's a good comparison since Philip Pfaffe claims it has the best performance on their evaluation[15].

4.7 Exact-Packed-String-Matching

Exact-Packed-String-Matching (EPSM) was presented by Faro and Külekc in 2013[16]. EPSM packing several characters into a bit-word based on the Intel streaming SIMD extensions and compared with a packed pattern bit-word. it uses compare idea like shift or algorithm, shift and bitwise-and operations are used to efficiently compare text chunks with the pattern. The asymptotic runtime of the algorithm is O(p*t), but in practice very fast.

5 Experimental Evaluation

In this section, I present the performance evaluation of parallel string matching algorithms. The sequential implementation of some codes comes from String Matching Algorithms Research Tool, then modified to accommodate multi-threading or GPU programming requirements.

Here are the datasets I used for comparison:

- A sample of natural language (the text of the Bible, 3.1MB)

- A set of genome sequences (consisting of four letters alphabet A,T,C,G, 98MB, 672MB, and 953MB)
- Extreme data generated by some programs (consisting of only two letters alphabet a,b and only 1 b,1GB)

In order to evaluate the benefits of parallelization, To eliminate any impact of I/O latencies, the input files were fully loaded into memory before running my algorithm, and then I tested my algorithm using 1, 2, 4, 8, 16, 32 threads, and their respective running times are recorded, and used test patterns with lengths of 2, 4, 8, 16, 32, 64 and 1000.
Due to the particularity of the GPU, the running time of the program on the GPU will include memory allocation and program running time but does not include the time for the results to be copied back.
All of the CPU experiments were conducted on an AMD ryzen9 3950x CPU, compiled with opencilk and linked to sse4.2. The CPU has 16 cores (32 hardware threads) clocked at 4.1GHz (overclocked). The GPU used was an NVidia RTX 2070 with 8GB of memory, running the latest version of CUDA (22.04)
In the following subsection, I discuss an excerpt of my result data.

Table 1: Pattern of natural language text (bible, each slice function runs in 32 threads)

Algorithm	2 Pat	4 Pat	8 Pat	16 Pat	32 Pat	64 Pat
Naive	0.042	0.044	0.045	0.045	0.045	0.044
Slice Naive	0.01	0.01	0.01	0.01	0.011	0.011
Parallel Naive	0.015	0.007	0.004	0.004	0.004	0.004
Boyer-Moore	0.1	0.066	0.036	0.021	0.013	0.009
Slice BM	0.01	0.007	0.007	0.006	0.004	0.004
KMPSearch	0.11	0.112	0.113	0.113	0.113	0.113
slice KMP	0.011	0.012	0.012	0.012	0.012	0.012
Shift Or	0.024	0.024	0.024	0.024	0.024	0.021
slice Shift Or	0.006	0.005	0.005	0.006	0.006	0.006
RabinKarp	0.042	0.034	0.03	0.03	0.03	0.03
slice RabinKarp	0.007	0.006	0.006	0.006	0.006	0.006
EPSM	0.003	0.003	0.003	0.004	0.002	0.002
slice EPSM	0.004	0.004	0.006	0.004	0.004	0.004
SSEF	0	0	0	0	0.09	0.092
slice SSEF	0	0	0	0	0.013	0.012
GPU Naive	0	0	0	0	0	0
GPU bm	0	0	0	0	0	0
GPU KMP	0.001	0.001	0.001	0.001	0.001	0.001
GPU SO	0	0	0	0	0	0
GPU RabinKarp	0.001	0.001	0.001	0.001	0.001	0.001

Table 2: Pattern of genome (953MB, each slice function runs in 32 threads)

Algorithm	2 Pat	4 Pat	8 Pat	16 Pat	32 Pat	64 Pat
Naive	2.047	2.229	2.24	2.248	2.243	2.243
Slice Naive	0.223	0.195	0.203	0.201	0.21	0.208
Parallel Naive	0.675	0.113	0.092	0.092	0.092	0.092
Boyer-Moore	3.461	2.684	2.04	1.449	1.098	0.478
Slice Boyer-Moore	0.233	0.195	0.183	0.154	0.162	0.142
KMPSearch	3.455	3.652	3.667	3.672	3.673	3.711
slice KMP	0.24	0.24	0.247	0.239	0.253	0.253
Shift Or	0.613	0.611	0.612	0.612	0.613	0.539
slice Shift Or	0.152	0.143	0.148	0.139	0.156	0.158
RabinKarp	1.176	0.861	0.756	0.745	0.746	0.749
slice RabinKarp	0.17	0.158	0.159	0.15	0.167	0.168
EPSM	0.078	0.084	0.167	0.094	0.057	0.044
slice EPSM	0.138	0.132	0.145	0.121	0.144	0.151
SSEF	0	0	0	0	2.219	2.294
slice SSEF	0	0	0	0	0.23	0.227
GPU Naive	0.113	0.111	0.117	0.252	0.173	0.227
GPU bm	0.373	0.265	0.22	0.207	0.197	0.186
GPU KMP	0.532	0.461	0.454	0.459	0.46	0.471
GPU SO	0.412	0.353	0.33	0.335	0.341	0.342
GPU RabinKarp	0.361	0.297	0.335	0.456	0.59	0.772

Table 3: Pattern of program generate text(each slice function runs in 32 threads)

Algorithm	4 Pat	8 Pat	16 Pat	32 Pat	64 Pat	1000 Pat
Naive	2.195	4.1	6.746	11.443	21.757	242.033
Slice Naive	0.31	0.396	0.58	0.936	1.591	16.428
Parallel Naive	0.136	0.246	0.399	0.754	1.424	16.395
Boyer-Moore	5.919	5.969	5.982	5.987	5.992	5.69
Slice Boyer-Moore	0.378	0.379	0.389	0.385	0.385	0.379
KMPSearch	4.866	4.909	4.914	4.916	4.913	4.764
slice KMP	0.342	0.355	0.363	0.364	0.362	0.355
Shift Or	0.803	0.805	0.807	0.808	21.928	241.736
slice Shift Or	0.213	0.218	0.228	0.219	1.655	16.468
RabinKarp	0.979	0.98	0.983	0.983	0.991	0.629
slice RabinKarp	0.234	0.232	0.245	0.243	0.242	0.235
EPSM	0.11	3.71	3.239	3.078	3.318	12.253
slice EPSM	0.198	0.514	0.481	0.442	0.495	1.714
SSEF	0	0	0	2.903	3.382	12.112
slice SSEF	0	0	0	0.328	0.37	
GPU Naive	159	261	187	325	297	2711
GPU bm	2130	2767	4054	7009	13327	701
GPU KMP	1664	1587	1591	1591	1606	1025
GPU SO	1604	1574	1577	1577	18104	226846
GPU RabinKarp	1942	2221	2926	4598	8282	780

5.1 Pattern

This is about the impact of patterns on parallel performance. When I want to analyze the benefits of threads, I want to find a relatively stable pattern as an input. Table 1, 2, 3 is the test result, in the test of natural language(Table 1) and genome(Table 2), it shows a short pattern (pattern less than 8) indeed has a certain impact on the algorithm, but when a pattern greater than 8, only the Boyer-Moore algorithm benefits from it, and the impact on other algorithms is marginal.
However, in the worst case, the effect of the pattern is very obvious. Brute force, Shift Or, and EPSM increase the running time significantly with the increase of patterns, while the KMP algorithm remains stable. This is because of the fact that EPSM partitions the text T into blocks to use bit parallelism compared to packed mode bit words. Due to the characteristics of memory comparison, the length of each comparison using bit parallelism is fixed, and the advantages cannot be utilized for large patterns, a disadvantage of using bit parallelism. The brute force algorithm needs to browse more patterns for comparison, while the shift or algorithm does not perform well in the case of pattern greater than 32. However, KMP and RabinKarp do not increase the running time because they can jump enough distance each time. Of course, in this case, the running time of the brute force algorithm reached 4 minutes, which is why cannot directly use the brute force algorithm, the running time, in this case, is unacceptable.

Table 4: thread of natural language text (bible)

Algorithm	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread	GPU
Naive	0.004	0.005	0.003	0.002	0.001	0.001	0
Boyer-Moore	0	0.001	0	0	0	0	0
KMPSearch	0.011	0.006	0.006	0.003	0.001	0.001	0.001
Shift Or	0.006	0.003	0.002	0.001	0	0	0
RabinKarp	0.003	0.001	0.001	0.001	0	0	0.001
EPSM	0	0	0	0	0	0	
SSEF	0.009	0.008	0.006	0.003	0.002	0.001	

Table 5: thread of natural language text (bible x 10)

Algorithm	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread
Naive	0.0446	0.049	0.028	0.027	0.016	0.011
Boyer-Moore	0.0088	0.007	0.007	0.005	0.005	0.004
KMPSearch	0.113	0.07	0.06	0.036	0.02	0.012
Shift Or	0.0212	0.025	0.02	0.012	0.008	0.006
RabinKarp	0.03	0.02	0.019	0.017	0.01	0.006
EPSM	0.0012	0.004	0.004	0.004	0.004	0.004
SSEF	0.0924	0.079	0.047	0.031	0.02	0.012

Table 6: thread of genome (953MB)

Algorithm	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread	GPU
Naive	2.239	1.38	0.862	0.482	0.301	0.208	0.227
Boyer-Moore	0.4774	0.437	0.269	0.2	0.16	0.142	0.186
KMPSearch	3.6798	2.293	1.201	0.666	0.387	0.253	0.471
Shift Or	0.5378	0.491	0.401	0.263	0.189	0.158	0.342
RabinKarp	0.7462	0.596	0.494	0.282	0.211	0.168	0.772
EPSM	0.0438	0.157	0.151	0.151	0.151	0.151	
SSEF	2.2944	1.363	0.985	0.549	0.339	0.227	

Table 7: thread of program generate text

Algorithm	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread	GPU
Naive	242.0744	122.293	121.241	61.643	31.869	16.428	2.711
Boyer-Moore	5.6868	3.127	1.799	0.98	0.562	0.379	0.701
KMPSearch	4.7578	2.788	1.593	0.877	0.515	0.355	1.025
Shift Or	241.949	121.963	122.396	63.411	31.811	16.468	226.846
RabinKarp	0.6234	0.779	0.508	0.363	0.287	0.235	0.78
EPSM	12.2426	1.709	1.711	1.711	1.733	1.714	
SSEF	12.1444	12.341	6.344	3.244	1.728		

5.2 Thread

This project focused on the impact of threads on the performance of different algorithms and the resulting speedup ratios. Table 4 shows the variation in running time of the matching algorithm on the Bible text under different numbers of threads, the pattern is 32. In the general text, because the running time is too short, it is difficult to show the specific time, so I copied the text ten times to observe more running time, which is in Table 5. In this case, most algorithms show a speedup under multi-core, except for EPSM. However, the sequential version of EPSM has a faster running time than the parallel version. The running speed on CUDA for some algorithms is slightly better than the sequential version, but it is difficult to exceed the parallel speedup of 16 threads.

Table 6 shows the variation in running time of the matching algorithm on the data for the genome file under different numbers of threads, the pattern is 32, where the alphabet of the genes leads to a large number of repeats in the small dataset ATCG. This causes an increase in the running time of the brute force algorithm and the KMP algorithm also performs poorly, while the other algorithms remain stable.

Table 7 shows the variation in running time of the matching algorithm on the data for the program-generated data under different numbers of threads, the pattern is 32. In this case, the brute force algorithm takes 4 minutes, while most algorithms take only seconds, with RabinKarp being the fastest at less than a second. In this case, I see a significant speedup, with all algorithms achieving a speedup of more than 5x in a 32-thread parallel. The time spent on Cuda is less than that of CPU parallel in this case, with the brute force algorithm taking only 2.7 seconds, a 90X speedup. This speedup ratio is much higher than that of all CPU parallel algorithms, indicating that under normal text, the GPU is unable to fully utilize its performance.

Table 8: Speed Up of natural language text (bible)

Algorithm	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread	GPU
Naive	1	0.8	1.333333	2	4	4	N/A
Boyer-Moore N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
KMPSearch	1	1.833333	1.833333	3.666667	11	11	11
Shift Or	1	2	3	6	N/A	N/A	N/A
RabinKarp	1	3	3	3	N/A	N/A	3
EPSM N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SSEF	1	1.125	1.5	3	4.5	9	

Table 9: Speed Up of natural language text (bible x 10)

Algorithm	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread
Naive	1	0.910204	1.592857	1.651852	2.7875	4.054545
Boyer-Moore	1	1.257143	1.257143	1.76	1.76	2.2
KMPSearch	1	1.614286	1.883333	3.138889	5.65	9.416667
Shift Or	1	0.848	1.06	1.766667	2.65	3.533333
RabinKarp	1	1.5	1.578947	1.764706	3	5
EPSM	1	0.3	0.3	0.3	0.3	0.3
SSEF	1	1.16962	1.965957	2.980645	4.62	7.7

Table 10: Speed Up of genome (953MB)

Algorithm	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread	GPU
Naive	1	1.622464	2.597448	4.645228	7.438538	10.76442	9.863436
Boyer-Moore	1	1.092449	1.774721	2.387	2.98375	3.361972	2.566667
KMPSearch	1	1.604797	3.063947	5.525225	9.508527	14.54466	7.812739
Shift Or	1	1.095316	1.341147	2.044867	2.845503	3.403797	1.572515
RabinKarp	1	1.252013	1.510526	2.646099	3.536493	4.441667	0.96658
EPSM	1	0.278981	0.290066	0.290066	0.290066	0.290066	
SSEF	1	1.683346	2.32934	4.179235	6.768142	10.10749	

Table 11: Speed Up of program generate text

Algorithm	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread	GPU
Naive	1	1.979462	1.996638	3.927038	7.595921	14.73548	89.2933973
Boyer-Moore	1	1.818612	3.161089	5.802857	10.11886	15.00475	8.11241084
KMPSearch	1	1.706528	2.986692	5.425086	9.238447	13.40225	4.6417561
Shift Or	1	1.98379	1.976772	3.815568	7.605828	14.69207	1.06657821
RabinKarp	1	0.800257	1.227165	1.717355	2.172125	2.652766	0.79923077
EPSM	1	7.163604	7.155231	7.155231	7.064397	7.142707	
SSEF	1	0.984069	1.914313	3.74365	7.028009		

5.3 Speed up

Analyzing which algorithm has a good speedup is one of the main purposes of this project. Table 8, 9, 10, 11 show the speedup ratios achieved in different types of data, respectively. Table 8 and 9 the variation in speedup ratios of the matching algorithm on the Bible text under different numbers of threads, the speed-up ratio is not high. On average, the speedup ratio can only reach less than half of the number of cores. While the KMP algorithm achieves a good speedup ratio still only 10X. And the brute force algorithm speeds up poorly, only achieving 4X speedup in 32 threads
Table 10 shows the variation in speedup ratios of the matching algorithm on the the data for the genome file under different numbers of threads,. The algorithms have achieved good speedup ratios except for EPSM, EPSM has brought extra time. KMP algorithm still achieved good speedup ratios, even 14X speedup ratio in 32 threads. Brute Force, SSEF, and Boyer-Moore have a speedup of 6-8X, while Shift-Or and Rabin-Karp are 4 and 5 times respectively. In the worst text shown in Table 11, the brute force algorithm has achieved a good speedup ratio, which is not without cost, and the time spent dropping from 4 minutes to 20 seconds is still unacceptable. In contrast, the speedup achieved by KMP and RabinKrap is far more than other algorithms, and have a good runtime. KMP even came to third place in speed in this case.

6 Conclusions

I implemented parallel versions of seven different string-matching algorithms by slicing the index according to the number of threads and compared their performance to the version on the GPU on different types and sizes of datasets. Most algorithms benefited from CPU parallelism, except for EPSM, with a 3-10x speedup observed in 16 threads. The brute force algorithms and KMP algorithms showed the greatest benefit from parallelism, KMP got 15X speedup in 32 threads. EPSM algorithms perform best on most natural texts, followed by Parallel brute force algorithms and Boyer-Moore algorithms. For large patterns, Rabin-Karp had the best performance. While EPSM had the best performance in genetics and generated text in a sequential setting, it did not benefit from parallelism except in the worst case. In the worst case, the GPU-parallelized brute force algorithm had a 90x speedup. In the general text, the algorithm had a 10x speedup on the GPU. I also observed the effect of the pattern size on the performance of the string-matching algorithms. For short patterns, EPSM had far superior performance to its parallel version, but it only the worst case seeing a speedup from parallelism. However, for 1000 patterns, EPSM is not good enough. This is because EPSM using bit parallelism only needs one instruction to compare within the range allowed by the memory, and the excess part still needs to be searched through a loop. In this case, Rabin-Karp and GPU-parallelized brute force algorithms were the clear winners.

Therefore, by analyzing the pattern, it is possible to select a better method for the string-matching problem. In the future, I plan to further analyze patterns and provide an optimal solution by choosing the most appropriate method for string comparison.

References

[1] Pfaffe, P., Tillmann, M., Lutteropp, S., Scheirle, B., Zerr, K. (2017). Parallel String Matching. In: , et al. Euro-Par 2016: Parallel Processing Workshops. Euro-Par 2016. Lecture Notes in Computer Science(), vol 10104. Springer, Cham. https://doi.org/10.1007/978-3-319-58943-5 15

[2] C. -L. Hung, T. -H. Hsu, H. -H. Wang and C. -Y. Lin, "A GPU-based Bit-Parallel Multiple Pattern Matching Algorithm," 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2018, pp. 1219-1222, doi: 10.1109/HPCC/SmartCity/DSS.2018.00205.

[3] G. Vasiliadis, M. Polychronakis and S. Ioannidis, "Parallelization and characterization of pattern matching using GPUs," 2011 IEEE International Symposium on Workload Characterization (IISWC), 2011, pp. 216-225, doi: 10.1109/IISWC.2011.6114181.

[4] Yuan Zu, Ming Yang, Zhonghu Xu, Lin Wang, Xin Tian, Kunyang Peng, and Qunfeng Dong. 2012. GPU-based NFA implementation for memory efficient high speed regular expression matching. SIGPLAN Not. 47, 8 (August 2012), 129–140. https://doi.org/10.1145/2370036.2145833

[5] Y. Liu, L. Guo, J. Li, M. Ren and K. Li, "Parallel Algorithms for Approximate String Matching with k Mismatches on CUDA," 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum, 2012, pp. 2414-2422, doi: 10.1109/IPDPSW.2012.298.

[6] A. Halaas, B. Svingen, M. Nedland, P. Saetrom, O. Snove and O. R. Birkeland, "A recursive MISD architecture for pattern matching," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 12, no. 7, pp. 727-734, July 2004, doi: 10.1109/TVLSI.2004.830918.

[7] Knuth, D. E., Morris, Jr, J. H., Pratt, V. R. (1977). Fast pattern matching in strings. SIAM journal on computing, 6(2), 323-350.

[8] P. D. Michailidis and K. G. Margaritis, "Performance Evaluation of Multiple Approximate String Matching Algorithms Implemented with MPI Paradigm in an Experimental Cluster Environment," 2008 Panhellenic Conference on Informatics, 2008, pp. 168-172, doi: 10.1109/PCI.2008.13.

[9] Someswararao, Chinta. (2012). Parallel Algorithms for String Matching Problem based on Butterfly Model. International Journal of Computer Science and Technology.

[10] A. Tumeo, O. Villa and D. G. Chavarria-Miranda, "Aho-Corasick String Matching on Shared and Distributed-Memory Parallel Architectures," in IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 3, pp. 436-443, March 2012, doi: 10.1109/TPDS.2011.181.

[11] H. Ishikawa et al., "PZLAST: an ultra-fast sequence similarity search tool implemented on a MIMD processor," 2021 Ninth International Symposium on Computing and Networking (CANDAR), 2021, pp. 102-107, doi: 10.1109/CANDAR53791.2021.00021.

[12] Knuth, Donald E., James H. Morris, Jr, and Vaughan R. Pratt. "Fast pattern matching in strings." SIAM journal on computing 6.2 (1977): 323-350.

[13] Karp, Richard M., and Michael O. Rabin. "Efficient randomized pattern-matching algorithms." IBM journal of research and development 31.2 (1987): 249-260.

[14] Boyer, Robert S., and J. Strother Moore. "A fast string searching algorithm." Communications of the ACM 20.10 (1977): 762-772.

[15] Külekci, M.. (2009). Filter Based Fast Matching of Long Patterns by Using SIMD Instructions.. Stringology. 118-128.

[16] Faro, Simone, and M. Oğuzhan Külekci. "Fast packed string matching for short patterns." 2013 Proceedings of the Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX). Society for Industrial and Applied Mathematics, 2013.

[17] Flynn's Classification of Computers. https://www.javatpoint.com/flynns-classification-of-computers

[18] KMP algorithm for pattern searching. GeeksforGeeks. (2022, December 1). Retrieved December 22, 2022, from https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/