15-618 Project Final Report: Parallel String Matching Algorithms

Abhishek Kumar and Runze Wang

Table of Contents

Summary

We implemented sequential and parallel versions of exact string matching algorithms: KMP, Rabin Karp, and Aho-Corasick. For parallel implementations, we used OpenMP and CUDA frameworks. We specifically targeted the scenario in which there are multiple patterns to be matched because of real-world applications in Network Intrusion Detection, Bio-informatics, and text mining. We ran our experiments on GHC machines, and we showed that CUDA and OpenMP implementations of Aho-Corasick significantly outperformed baselines like KMP and Rabin-Karp. Moreover, using parallelizing techniques CUDA and OpenMP is beneficial for the application of all three String Matching algorithms on large real-world datasets.

Background

KMP

Knuth-Morris-Pratt (KMP) algorithm is a linear-time searching algorithm that, given a pattern and text, finds all occurrences of the pattern in the text [1]. If the length of the text is *n* and *m* is the length of the pattern, then KMP works in *O(n+m)* time. It preprocesses the pattern to create a “longest prefix that is also a suffix” (LPS) array. In some texts, this array is referred to as the “failure function.” Thus, when matching the pattern, if there is a mismatch, it does not start from the beginning of the pattern but instead uses the LPS array to use the previous match information to continue matching from the middle of the pattern. Hence, it gets better worst-case complexity than the naive string search algorithm, which has *O(nm)* complexity in the worst case. It works on only one pattern at a time.

Rabin-Karp

Rabin Karp is a string search algorithm that uses hashing to find exact matches of patterns in the text [2]. It preprocesses the pattern to find its hash and then uses a rolling hash in the text. If the hash of the substring of the text matches the hash of the pattern, then it matches the pattern and the substring of the text character by character to confirm a full match. The worst-case time complexity can be as bad as the naive string search due to false hash matches. However, the expected runtime is linear, and this algorithm is widely used due to its simplicity. Like KMP, it works on only one pattern at a time.

Aho-Corasick

The Aho-Corasick algorithm [3] is used to find matching substrings for a set of patterns in a large input string. For example, given a dictionary of patterns *{a, ab, bab, bc, bca, c, caa}* and input text *abracadabra*, it outputs *a* occurs at indexes *0, 3, 5, 7, 10, ab* occurs at *0, 7, bab* doesn't occur anywhere,.... and so on for other patterns.

Aho-Corasick works on multiple patterns. The first step of the algorithm is to create a Finite State Automaton based on the Trie storing the patterns but with additional links that help faster transition between states when matching fails. The failure links behave similarly to the failure function of KMP, and in fact, Aho-Corasick is a generalization of KMP to multiple patterns. Furthermore, at each node of the Trie, there is an additional field called “output function”, which maintains the list of patterns that are matched at the given node. While algorithmically, the output function is described as a set, the output functions are represented as linked lists to support efficient unions. When we insert a pattern string to Trie, we insert a linked list node to the output function at the node corresponding to the end of the pattern. When we build up the failure links, we will merge the linked lists representing output functions at nodes, which can be done by connecting the end of the linked list at the current node to the output function at a previous node.

Aho-Coarsick is used in the Unix command *fgrep*. It is used in various applications like network intrusion detection and bioinformatics for finding several input strings within a given large input string. Suppose n is the length of the text and m is the total number of characters in the search dictionary. In that case, the Aho-Corasick algorithm works in $O(n+m+z)$ complexity, where z is the total number of occurrences in the text.

Application of String Matching Algorithms

String matching algorithms have a wide gamut of applications. Information retrieval, molecular biology, data compression techniques, text mining, plagiarism detection, digital forensics, and network intrusion detection systems - all use string-matching algorithms on large amounts of data [4]. Traditionally, these algorithms have been run sequentially on the CPU. However, with astronomical growth in the size of data used in these applications, the sequential versions of the algorithms have proved insufficient. The advent of parallel computing (especially heterogeneous computing) has paved the way for a lot of techniques that have explored running parallel versions of these algorithms on heterogeneous architectures like general-purpose GPU [5] [6] [7].

Network intrusion detection systems (NIDS) use string-matching engine to identify malicious content in network packets by searching for thousands of predefined patterns. NIDS are critical to protecting from Denial of Service attacks and port scans [5]. The string search engine is usually the performance bottleneck for NIDS.

In this work, we look at two major parallelizing techniques. We use OpenMP to parallelize KMP, Rabin Karp, and Aho-Corasick algorithms. OpenMP is a shared memory parallel programming technique for C/C++ and Fortran. The API is very simple, portable, and scalable, and it is useful for developing parallel applications on platforms from the desktop to the supercomputer having a multi-core CPU. We also use CUDA to parallelize these three algorithms to run them on general-purpose GPUs.

Approach

Parallelizing Aho-Corasick

Parallelizing Aho-Corasick via OpenMP

The OpenMP implementation of Aho-Corasick aims at parallelizing three stages of applying Aho-Corasick for matching against multiple patterns: constructing the Trie storing the patterns, constructing the failure links using BFS, and match the text against the patterns. While the algorithm itself will be similar to the sequential algorithm, parallelizing the three stages of Aho-Corasick requires a slightly different set of tools.

Parallel Construction of Trie via OpenMP

Recall that to insert a pattern into the Trie, one simply starts from the root node of the Trie and moves down along the path corresponding to the pattern string. If there is no corresponding node, we will insert a new node to Trie. This process is inherently sequential. Since we have multiple pattern strings, one natural idea is to insert the pattern strings in parallel, and each thread is responsible for inserting a subset of pattern strings into the Trie. This means we would need to implement lock-free insertion algorithms for Trie.

Trie insertion can be made lock-free using compare-and-swap primitives. Notice that a new Trie node would always have the same metadata: we don't keep track of the parents of nodes in Trie. This enables us to create a new Trie node at the start of the iteration. When we try to go down along the Trie, we use compare-and-swap to atomically check if there are no nodes corresponding to our next step. If there are currently no nodes, we will insert the node to the Trie. This logic is exactly the same as the logic we would use for the sequential implementation, and since no deletions are involved, then we don't have to worry much about the ABA problem. The algorithm for a lock-free insertion looks like this:

```
ParallelTrieInsert(string pattern):
  current = (root of Trie)
  newNode = new TrieNode(alphabetSize)
  for (ch in pattern){
      if ch is not in the alphabet:
            throw error

      //Use compare-and-swap to check if there are no edges labeled “ch”.
      //If so, use the new node and create such an edge.
      flg = __sync_bool_compare_and_swap(&(current->child[ch]), NULL,
newNode)
      if (flg):
          Increment trieSize atomically
          newNode = new TrieNode(alphabetSize)
      //In any case, follow the edge labeled “ch”.
```

```
      current = current->child[ch];

    // Similarly, insert the output node for the word using
    // lock-free linked list insertion.
    outputNode = new LinkedListNode;
    outputNode->val = length;
    outputNode->next = NULL;
    // Try and see if current linked list is empty
    flg = __sync_bool_compare_and_swap(&(current->outputTail), NULL,
outputNode);
    if (flg):
      // This means outputTail was NULL, so we have an empty linked list
      current->outputHead = outputNode
    else:
      // Try to insert the node at the end of the linked list.
      while (1):
            oldTail = current->outputTail;
            if (__sync_bool_compare_and_swap(&(current->outputTail),
oldTail, outputNode)):
                  oldTail->next = outputNode
                  break
    delete newNode
```

Finally, recall that we use linked lists to store the “output set” corresponding to Trie nodes, which correspond to the set of patterns matched if we end up with the given node of the Trie. If there are no duplicate patterns, then each thread will end up with different final nodes corresponding to the patterns, and there are no race conditions involved: each thread can just create the node in the linked lists independently. On the other hand, if we allow duplicate patterns and we count matches against duplicate patterns multiple times, this would require us to use lock-free linked list insertion algorithms. We chose the later approach for greater generality. If we only have insertions, then lock-free linked list insertion is similar to inserting nodes to Trie. We can simply use compare-and-swap to check if we are inserting after the current tail. If so, we reset the tail to the new node and add the “next” link from the old tail to the new node. If not (due to another thread updating the tail of the linked list first), we will just retry. We have tested our implementation against duplicate patterns to make sure our link-free insertion mechanism is working.

Parallel Construction of Failure Links via OpenMP

The sequential implementation of Aho-Corasick creates failure links via BFS (Breadth-first search). The natural way to parallelize the construction of failure links is via

parallel BFS. In particular, since we are dealing with a tree data structure instead of general graphs, there are fewer places where race conditions can become a barrier: we will only be able to discover a node from its parent in Trie, so it is not possible for two different threads to discover the same node and compute data (failure links and output functions) related to the node. The difference between parallel BFS and sequential BFS is instead of maintaining a single queue for the list of nodes we are going to visit, the parallel BFS implementation would maintain the "current frontier" and "next frontier". The frontier would correspond to nodes in the Trie with the same depth. Initially, the "current frontier" is just the nodes on the level immediately before the root node (which corresponds to the first letters of the patterns). In each iteration, the threads will process the nodes in the current frontier in parallel to find out the next frontier. This means exploring all nodes from the a node in the current frontier and compute the corresponding failure links and output functions for those new nodes. In the end of the iteration, we set the computed next frontier as the new current frontier.

Since we are doing BFS over a tree instead of a general graph, the only place we need to worry about data races is placing a newly discovered node into the frontier. Nevertheless, we can maintain a counter to denote the nodes currently in the new frontier. When we want to insert a node to the frontier, we simply atomically capture the current counter and increment the counter by 1, then insert the node to the corresponding index.

The pseudocode for parallel BFS over the Trie can be described below:

```
computeFailPointersParallel()
    currentFrontier, nextFrontier = []
    Insert initial nodes to the current frontier.
    while (currentFrontier is not empty) do
      nextFrontier = []
      parallel for nodes u in currentFrontier:
          for nodes s from node u:
            Insert s into nextFrontier (lock free).
            Compute the fail pointers and output functions.
          end
      end
      currentFrontier = nextFrontier.
    end
```

Finally, it should be remarked that simply changing all for loops in a typical sequential BFS implementation (which maintains a queue for the nodes) won't work: since originally, the for loops only iterate through outgoing edges from a node, this means the for loop will only iterate through the alphabet. Since alphabet size is usually small, we are not generating enough parallel tasks. We also need to make sure the enqueue operation is protected by a lock, so naively replacing all sequential for loops with OpenMP `#pragma omp parallel for` does not work, and results in much poorer performance than a sequential implementation.

Parallelizing Matching Phase via OpenMP

Our OpenMP implementations of KMP, Aho-Corasick, and Rabin-Karp follow a similar approach for matching pattern (KMP, Rabin-Karp) or patterns (Aho-Corasick) against the text:

namely we divide the text into chunks of roughly equal size: each thread is responsible for approximately the same amount of possible starting positions for the matches. This approach means we also need to pad characters so that if we start the matching at the last character of a block, we can still match the text against the longest possible pattern. Since KMP, Aho-Corasick, and Rabin-Karp perform roughly the same amount of work at each character (though the actual cost of following failure links for KMP and Aho-Corasick might fluctuate), we decided to implement a static workload assignment strategy.

When testing our initial implementation of OpenMP Aho-Corasick against patterns where there are very frequent matches due to duplicate short patterns, we observed that the performance deteriorated significantly in the sense that it performs even worse than the sequential implementation. Our solution was instead of maintaining a shared counter for occurrences, each thread is responsible for a different counter and the counters are padded to make sure there are no false sharing. When threads finish matching, we simply use OpenMP reduction primitive to sum up the counters. While this optimization isn't relevant if matches are rare, if matches are very common, this can lead to potential great speedup due to less coherence traffic and no need to make atomic updates to a shared counter.

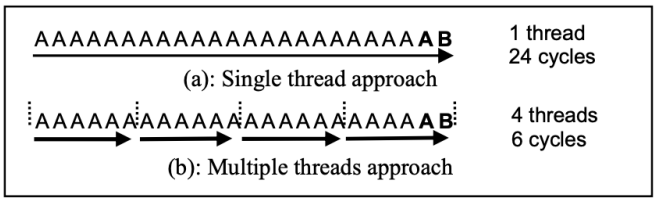
Parallelizing Aho-Corasick via CUDA

If we want to implement the Aho-Corasick algorithm over CUDA, we need to consider additional constraints like CUDA threads are executed in warps. If we have divergent CUDA threads, we will end up with poor performance. The program model of CUDA also makes it difficult to implement parallel construction of Aho-Corasick automaton, since lock-free Trie insertion and parallel BFS are not exactly perfect fits for data parallelism. As a result, we focused on parallelizing only the match phase via CUDA.

The approach we took was the approach used by Lin et al. in [5] and by Thambawita et al. in [7]: instead of dividing the text into too many segments and running into high padding overhead or divergent control flows, we just neglect the failure function for the moment. Each thread is responsible for a possible start position: starting from the given position, it tries to go down along the Trie and try to match the text against the pattern. This approach is called "parallel failureless-AC algorithm" in [5] and in [7].

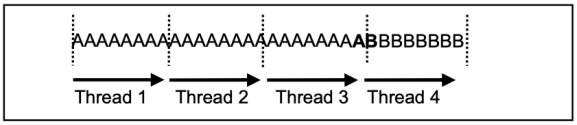
Parallelizing KMP and Rabin-Karp

Both KMP and Rabin Karp algorithms are inherently sequential are difficult to parallelize algorithmically. However, there are two potential axes of parallelism. The first is parallelizing over patterns and the second is dividing the text data in chunks. We used both axes of parallelism.



Single vs. multiple thread approach [1]

For OpenMP implementation, we first parallelized over patterns and then parallelized over chunks of data. Chunking of data leads to boundary problem as shown in the below figure. Pattern AB can't be detected by either Thread 3 or Thread 4. We solved this problem by letting threads do overlapped computation on boundaries.



[2]

Implementing KMP and Rabin Karp using CUDA was a bit more involved. We copy the data structures - LPS array for KMP and Pattern Hashes for Rabin Karp to CUDA global memory. We used patterns as the first axis of parallelism. That is, thread 0 in block 0 is responsible for pattern 0 on the first text chunk, thread 1 in block 0 is responsible for pattern 1 on the first text chunk…..and so on. This helps in data locality because all threads in a warp will be working with the same text chunk (assuming we have greater than 32 patterns). We also need to handle boundary problem here and we do that by letting threads do overlapped computation on the boundary.

One possible optimization to improve the performance of CUDA implementations could be pipelining the data transfer and computation. Instead of copying the whole large text to CUDA global memory in the beginning, the text can be divided into streams. One stream can be copied to GPU memory while computation is going on the previously copied stream. In this way, we will overlap computation with communication and it should significantly increase the performance of the CUDA implementation. We did not implement this due to time restrictions.

Theoretical Analysis

When there is only one pattern p and a text s, then as we have seen in the "background" section, the asymptotic runtime of Aho-Corasick, KMP, and Rabin-Karp are all $O(|p| + |s|)$. Nevertheless, Rabin-Karp has regular memory access patterns but carries expensive instructions like multiplication and division. KMP and Aho-Corasick may require us to move along the failure pointers if the text does not match the pattern, which would create some overhead. Moreover, the branching behavior of KMP and Aho-Corasick, as we shall see in experiments, can lead to poor performance on actual machines since missing branch predictions are costly on modern CPU architectures. Therefore, even though they have the

[1] Image Credit [5]
[2] Image Credit [5]

same asymptotic time complexity when there is only one single pattern, they can have very different performance on actual machines.

If there are multiple patterns $p_{1,}..., p_m$, then the runtime of Aho-Corasick is given by $O(|s| + \sum |p_i| + z)$ where z is the total number of matches in the text. In most applications, z is way smaller than $m|s|$: it is usually not the case that most starting positions will match with most of the patterns. Of course, if say the patterns are all character 'a's and the text consists solely of 'a's, then $z = m|s|$, but most of the time we don't have to worry about these types of degenerate scenarios and we care most about the $O(|s| + \sum |p_i|)$ term.

On the other hand, KMP and Rabin-Karp are both designed for single patterns, so the runtime would scale with the number of patterns. Both algorithms have a asymptotic time complexity of $O(m|s| + \sum |p_i|)$, and we have to go over the text m times. This behavior means it is usually a bad choice to choose KMP and Rabin-Karp when we have multiple patterns.

As we shall see soon, the theoretical analysis does not take into account factors like branch predictor misses, prefetching, cache misses, and SIMD. These factors would contribute to wildly varying actual run time.

Results

Benchmarks on Generated Data

In this section, we are going to explore experiment results over different types of computer generated test data. We are running in the "find all occurrences" mode, which means that the matching algorithms must report all matches against the patterns instead of simply counting the patterns. We design the number of matches to be small so that the sequential part of inserting matches to the data structure does not block the overall progress of parallel algorithms. We use GHC machines for running our experiments.

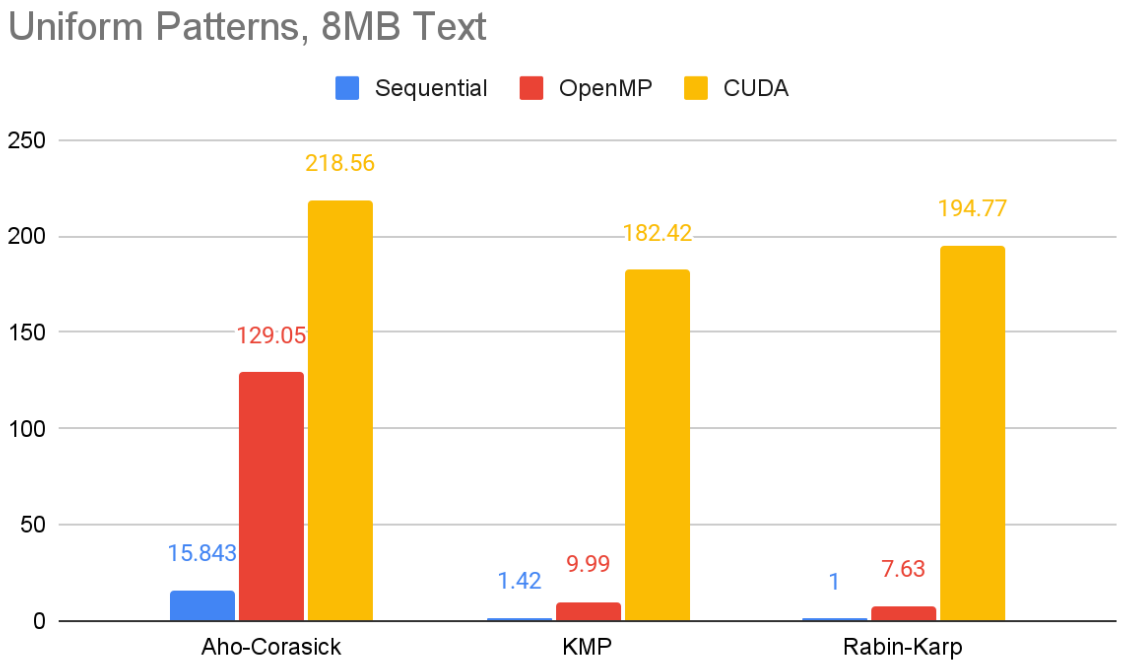
Case 1: Short, Uniformly Random Patterns and Long Text

The first case we consider is matching a long text against some short, uniformly random patterns. Here, "uniformly random" simply means the text and the patterns are constructed by sampling each character from the alphabet uniformly randomly and independently. Since we still hope that we end up with some matches, if the text length is $|s|$ and we want to have m patterns, the probability that a starting position matches the pattern is simply $|\Sigma|^{-|p|}$. Thus, we will try to make sure $|s| \approx m|\Sigma|^{|p|}$. Then we can solve for pattern length $|p|$.

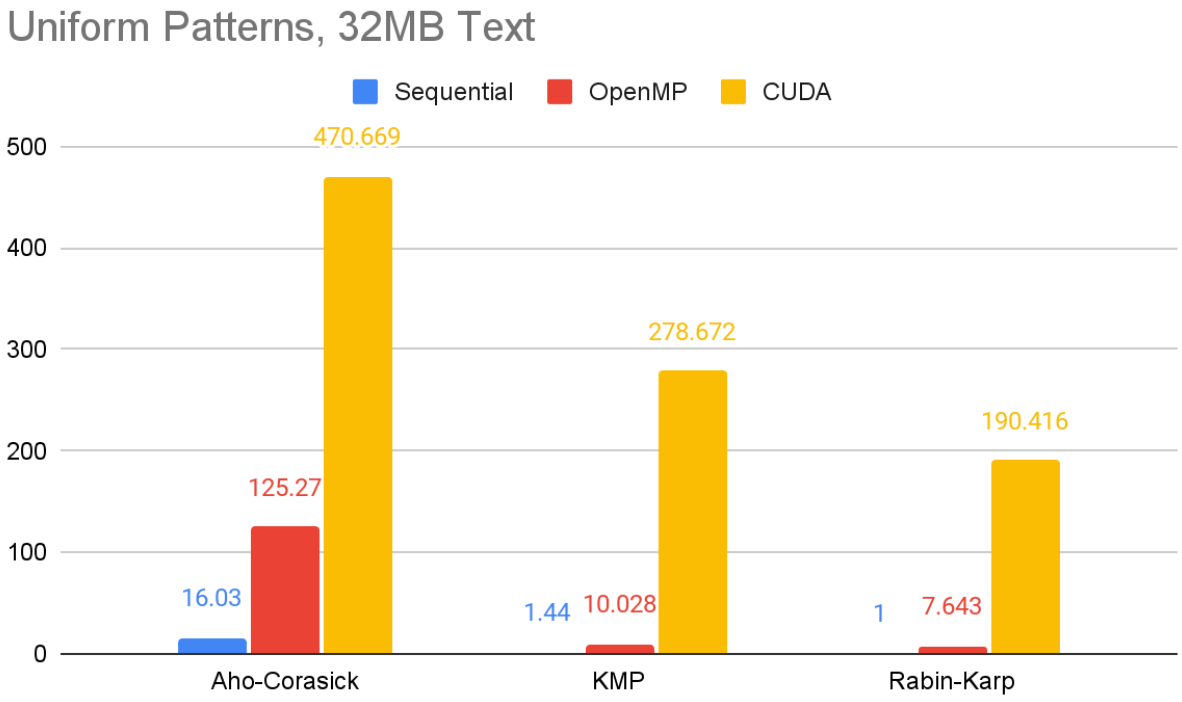
Since GHC machines operate on Core i7-9700, which has 12MB of L3 cache, the text sizes we consider are 8MB and 32MB, respectively. We picked $|\Sigma| = 4$ (which correspond to the alphabet size of DNAs and to make m larger). We chose $m = 24$ to make sure that we don't

have too many patterns for KMP and Rabin Karp but the number of patterns divide the number of cores, and we chose pattern length $|p| = 15$.

This setting is designed to be a relatively easy task since the pattern lengths are uniform and strings are generated uniformly randomly. We expect that starting from a given position of the text, on average we won't be matching a lot of characters with parting. Here are the experiment results over speedups over the sequential Rabin-Karp implementation:



Uniformly random patterns and text, 8MB Text.



Uniformly random patterns and text, 32MB Text.

Furthermore, our experiments show that the runtime of sequential Rabin-Karp scaled roughly linearly with respect to input size: the runtime of sequential Rabin-Karp is 1.07s over 8MB text and 4.30s over 32MB text. In fact, the sequential implementation of Rabin-Karp always takes approximately 1.07s to finish on a text with size 8MB, assuming the sizes of the patterns are not excessively large. On the other hand, as we expect, since 32MB text does not fit into L3 cache, running over 32MB text incurs a much higher cache miss rate. Here are the miss statistics for sequential Rabin-Karp over 32MB text and 8MB text:

Text Size	Cache References	Cache Misses	Miss Rate
8MB	6,285,632	581,874	9.257%
32MB	25,129,577	13,397,849	53.315%
Ratio	3.998	23.025	5.759

Nevertheless, since our access pattern of the text is highly regular (we read the text in order), even though extremely long texts may result in worst cache miss rates, these cache misses can be easily predicted and prefetching can help to hide the cost of cache misses. As a result, we see a roughly linear blowup when the text size grows from 8MB to 32MB.

One immediate observation is that the speedup of Aho-Corasick over KMP is not proportional to the number of patterns, even though Aho-Corasick is equivalent to KMP if there

is only one pattern string. Since Aho-Corasick transitions are stored in a Trie instead of an array, Aho-Corasick inherently suffers from a less compact representation of data. In particular, with a pattern length = 15, the KMP LPS array for a given pattern can fit into one cache block, but the entire Trie cannot fit into one cache block. Therefore, if after some steps, we end up at a node that is not in the cache because we rarely visit the node, we might incur a cache miss. This cache miss can be hardly hidden by prefetching since matching along the automaton is a "pointer chasing" process. The cache miss statistics for sequential implementations of Aho-Corasick and KMP when there is only one pattern (while keeping other parameters of the experiments) makes the issue clearer:

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

Indeed, most of the cache misses here are cold misses when we load the text into the memory: a 8MB text corresponds to 125,000 cache blocks (assuming cache blocks have size 64 bytes), and allocating an 8MB space of the heap (as implemented by `std::vector`) and filling the text with random characters seems to be giving us a 2x overhead here. Nevertheless, we can see that sequential AC have significantly more cache misses than sequential KMP. The figure is more significant if you subtract the approximately 250,000 cold misses. As a result, less compact representation of Aho-Corasick automaton (which does not fit into one cache block) results in higher cache miss rates during the matching process and result in poorer performance compared to KMP when there is only one pattern string.

Another immediate observation is that the speedup of most algorithms over sequential Rabin-Karp is consistent over 8MB text and 32 MB text, except the Aho-Corasick implementation with CUDA and KMP. First, since our initial experiments ask for finding all matches and CUDA does not have dynamic global memory on the device side, we have to allocate a sufficient amount of memory for all matches beforehand and communicate between the host side and device side about all matches. This contributes to a high memory overhead for CUDA implementations of Aho-Coarsick and KMP. In fact, if we set the experiment to "count matches only", then we see more significant speedups for the CUDA implementations of Aho-Corasick and KMP, and the performance over 8MB text and 32MB text is more consistent. Furthermore, in this case, matching against 32MB text using the CUDA implementation of Aho-Corasick only requires 4.485 milliseconds counting the time required to transfer text to GPU side. Recall that from Assignment 2 we learned that 16x lane PCI 3.0 has a theoretical peak transfer time of 15.7GB/s, we can immediately conclude that the theoretical minimum time required to transfer 32MB data to GPU side is approximately 2 milliseconds. As a result, we conclude that CUDA Aho-Corasick is mainly bound by the overhead of data transfer between host side and device side.

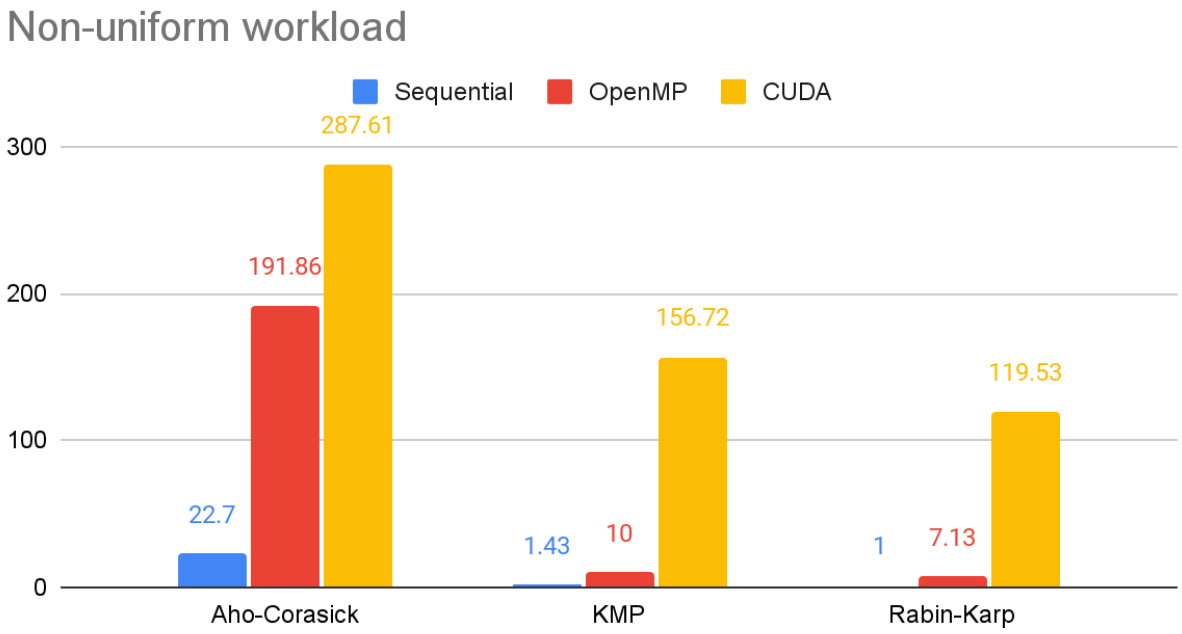
Besides these observations, we can see that in general, Rabin-Karp is slower than KMP, which would make intuitive sense given that Rabin-Karp requires slow arithmetic operations like multiplication and division. The OpenMP matching construction achieves almost perfect

speedups for all three algorithms, which is expected: first, the cost of matching a given block of text against patterns is uniform. Second, since the number of patterns divides the number of cores, if we loop over the patterns in parallel in OpenMP implementations of Rabin-Karp and KMP, each thread will be responsible for roughly the same amount of patterns.

Case 2: Workload Imbalance

In this case, we aim at measuring the effects of non-uniform pattern lengths and a pattern number that does not divide the total number of cores. In Aho-Corasick, KMP, and Rabin-Karp, the pattern lengths have an additive impact on the runtime instead of a multiplicative impact (multiplicative impact means when the pattern lengths grow by a factor of 2, the runtime also grow by a factor of 2). Furthermore, we implemented parallelism over the text, which means we divide up the text into smaller blocks. These reasons imply that theoretically, the threads should have a balanced workload even if the pattern lengths differ and the number of cores doesn't always divide the number of patterns.

We set up the experiments as follows: we generate 36 patterns. 4 of the patterns have length 5, while the rest of the patterns have length 1000. Notice that this time the number of patterns does not divide the number of cores so that when we loop over the patterns in parallel, we will encounter some workload imbalance. This setup can be challenging for OpenMP implementations of Rabin-Karp and KMP. Nevertheless, we have enough patterns to generate sufficiently many parallel tasks, counting the subtasks of matching a pattern against a block of text. Also, patterns having length 5 means we will encounter matches quite often, and thus matching those patterns against the text would involve slightly more work than matching longer patterns that makes having a match almost impossible. Since we protect the list of matches with a lock, we don't want to have matches so frequent that the sequential part of the algorithm (i.e. adding matches to the list of matches) dominates the runtime. The text size is still 8MB:



As we might expect, since our primary axis of parallelism for OpenMP implementations of KMP and Rabin-Karp are patterns, when the number of cores doesn't divide the number of patterns, we will be experiencing some workload imbalance. Nevertheless, since we further generate parallel subtasks of matching against portions of the text and OpenMP can support nested parallel regions, overall we still get close to 7x speedup compared to the sequential implementations of KMP and Rabin-Karp.

Case 3: Long Matches

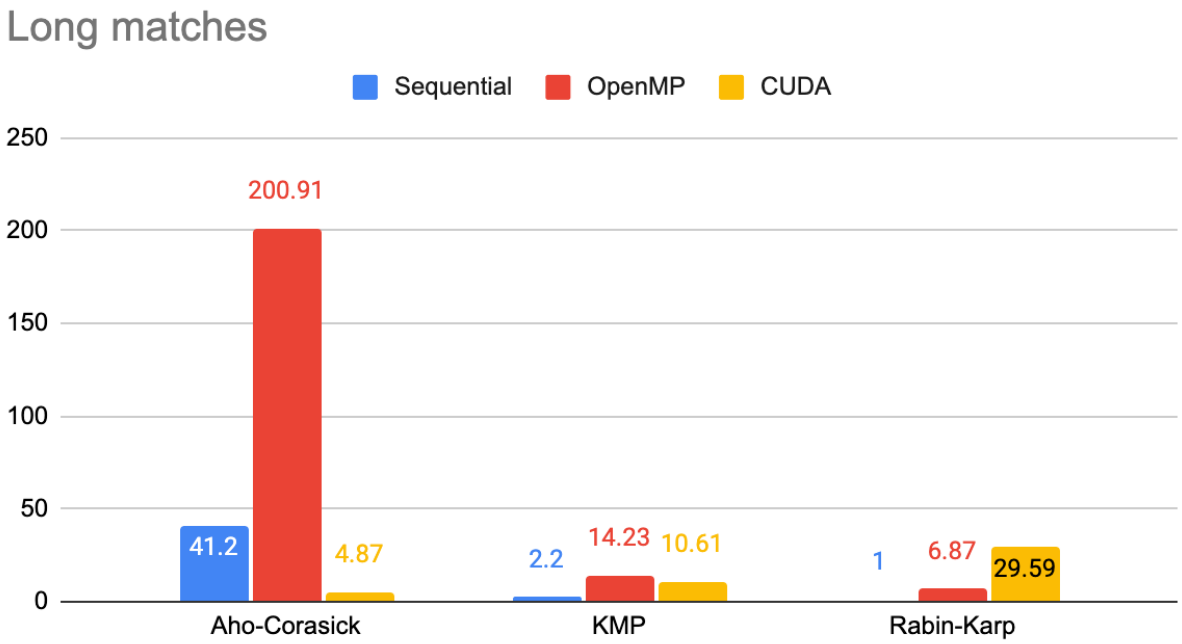
If the pattern and the text are sampled uniformly randomly and independently from the set of strings over a non-trivial alphabet with given length, then brute force matching will work very well since on average, the matching process stops quickly. On the other hand, if at most of the positions, we can match most of the pattern, then the brute force will spend a lot of time trying to match the text starting at a given position against the patterns. Consider the case where the text string contains $2L$ letter "a"s and the pattern string contains L letter "a"s. In this degenerate example, there pattern would match the text from $L + 1$ starting positions, and if we sum up the matching lengths from the given position, it would be

$$\sum_{i=1}^{L} min(L, L + 1 - i) = O(L^2).$$

Having many long matches is obviously not ideal for brute force matching, and this is where algorithms like KMP and Rabin-Karp exhibit great speedup against brute force algorithms.

To generate a test case where there are many long matches, we first create the building blocks of our text and our patterns: the building block consists of many repeated characters at

the front plus a random short suffix. The patterns consist of single building blocks, while the text consists of random building blocks concatenated together. This would make most starting positions of the text match the pattern for extended lengths until we enter the random suffix regions. We keep the text size to be 8MB. Each building block consists of 9992 ‘a’s plus a random suffix with length 8. There will be 24 such patterns, and alphabet size is still 4.



The experimental results turn out to be tragic for CUDA implementations of Aho-Corasick and KMP: the CUDA implementation of KMP is slower than the OpenMP implementation of KMP, and the CUDA implementation of Aho-Corasick is even slower than the sequential implementation of Aho-Corasick. Nevertheless, since we have extremely long patterns (compared to the text), if we try to parallelize the matching phase by dividing the text into equal trunks, there will be non-negligible cost for paddings. Algorithms over CUDA only work well if we can have many fine-grained threads that fit a SPMD paradigm, so the CUDA implementation of Aho-Corasick would give up the failure links. Nevertheless, when there are many long matches, giving up failure links means starting from a given position of the text, we can expect to match against the pattern for extended time until entering the random suffices that may terminate the matching process. Our conclusion here is that these adversarial test cases mean an efficient CUDA implementation against all possible test cases is highly nontrivial.

Under this scenario, we see that (sequential) KMP have even more advantage compared to Rabin-Karp. If we take a close look at the implementation of Rabin-Karp, we see that when we have long pattern matches, we have poor locality since our access pattern to the text is `text[i], text[i+|p|], text[i+1], text[i+|p|+1]...` This access pattern is not cache friendly but predictable, so the runtime of sequential Rabin-Karp is still around 1.07 seconds. In the meantime, if KMP is able to match from a position, then it will just behave like brute force: it just iterate through the pattern and through the text with increment of

1, until finding a match or finding a mismatch (which would require us to move along the failure function). This means if we have long matches, KMP should have especially good spatial locality.

Finally, we noticed that the OpenMP implementation of Aho-Corasick is only able to achieve a 5x speedup against the sequential implementation of Aho-Corasick. If we scale the block length to 1000 (i.e. a prefix of 992 characters and a random suffix of 8 characters) and keep the text size at 8MB, the speedup is approximately 7.15x over the sequential implementation of OpenMP. Again, we chose to run cache reference statistics:

Scenario	Cache References	Cache Misses	Miss Rate
|p| = 1000, seq.	710,354	278,106	39.150%
|p| = 10000, seq.	1,528,486	306,099	20.026%
|p| = 1000, OpenMP	2,941,851	298,412	10.144%
|p| = 10000, OpenMP	20,060,489	386,081	1.925%

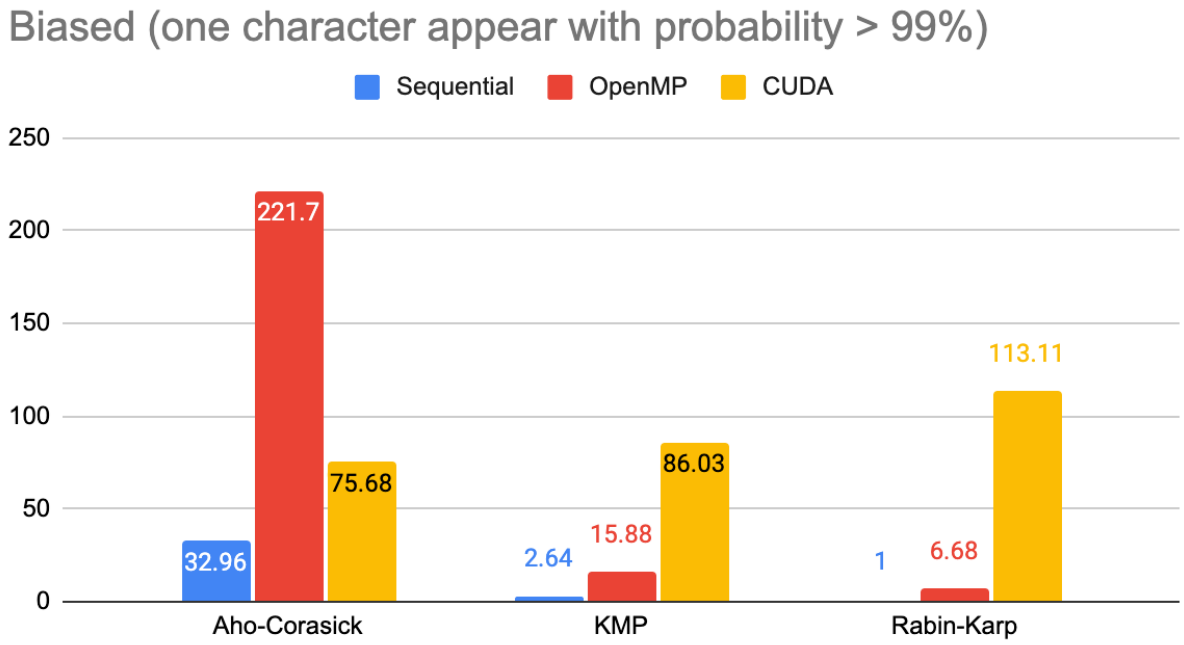
Recall that generating and loading a text of 8MB will give us approximately 250,000 cold cache misses. In the meantime, we see that OpenMP implementation will make much more cache references than the sequential implementation. Upon further investigation, it occurs that the sequential implementation of Aho-Corasick stores the nodes of Trie in a vector. In particular, in this “long match” scenario, we simply start from the root and move down along the Trie since the building blocks of the text and the patterns have long prefixes consisting of the same character. If we insert the pattern strings in order, then the nodes of the Trie corresponding to the common prefix of patterns would be stored together (i.e. node with ID 1 correspond to ‘a’, node with ID 2 correspond to ‘aa’, node with ID 3 correspond to ‘aaa’, etc.), and our access pattern along the Trie when we match the text against the patterns will have good spatial locality since most of the time (i.e. when we enter a long repeat of the prefix character) we are moving down the Trie and accessing the Trie nodes in order . On the other hand, our OpenMP implementation uses pointers to store the nodes of Trie to enable lock-free insertion. This means that if we walk along the Trie, the nodes are not guaranteed to be close spatially in memory. We should also recall the fact that a Trie node needs to store way more information: failure links, output functions, and pointers to the children. While the Trie in this case can still fit inside the L3 cache, the poor access locality implies caching is not as efficient. This is why we are only seeing a 5x speedup of the OpenMP implementation of Aho-Corasick against the sequential version.

Case 4: Skewed Distribution of Characters

In natural languages like English, the distribution of letters is heavily skewed. Letters like “e” or “t” occur much more frequently than letters like “x” or “z”. As a result, it is worth studying what will happen if the distribution of character frequency in the text and in the pattern is heavily skewed.

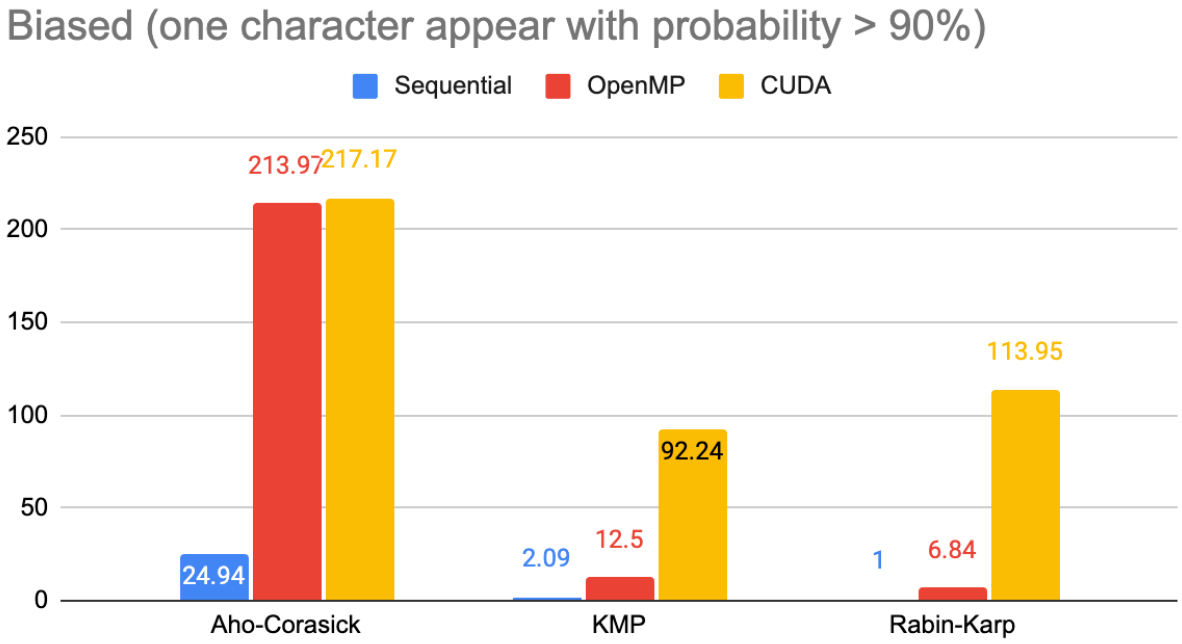
There are two subcases: whether one character appears significantly more often than others or there are multiple characters appearing significantly more often than others. If only one character appears significantly more often than others, we will end up with a case similar to the previous “long match case”: we can still expect that there will be many long partial matches. If there are multiple characters appearing more often than others, then we do not necessarily have long partial matches. Nevertheless, a skewed distribution of characters means we may face different match and access patterns.

In this experiment, we first tried the case where most of the characters of the pattern and of the text are ‘a’s. In particular, the patterns and the text are generated as follows: with 99% probability, the character is an ‘a’. With 1% probability, it would be uniformly random among ‘a’,’b’,’c’, and ‘d’. The characters in the patterns and in the text are sampled independently. The pattern length is now 1000 (since this means we have close to 10 spots that are randomly generated among ‘abcd’). There are still 24 patterns, and the text size is still 8MB. The experiment result is presented below:

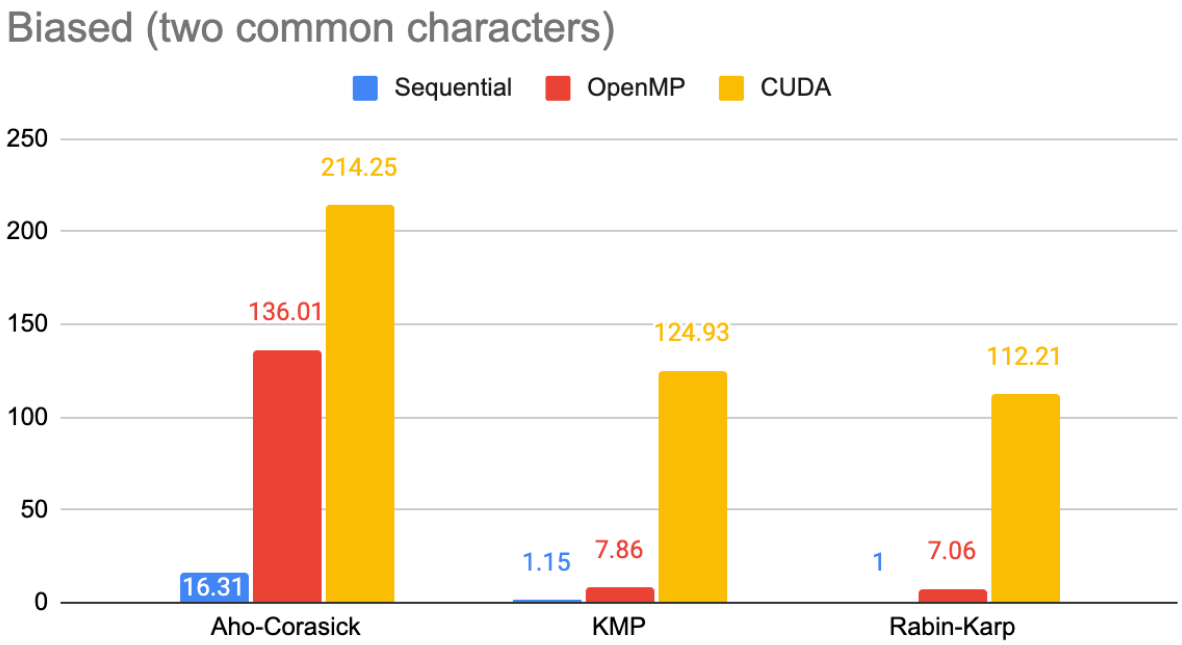


As one might expect, the patterns are long and starting from a given position, on average we can expect we are able to match the pattern for an extended period. Since the pattern lengths are shorter, we witness a less extreme version of the “extremely long matches” we constructed before.

What if we change the setup to “with 90% probability, we pick ‘a’, and with 10% probability, we pick uniformly randomly from ‘a’, ‘b’, ‘c’, and ‘d’?” We can expect that the expected match length from a given position will be greatly reduced, so CUDA implementation of Aho-Corasick will perform better. And the experiments show that it is indeed the case:



One interesting phenomenon we have observed is that this case is particularly good for the OpenMP implementation of Aho-Corasick, but not so for the OpenMP implementation of KMP. We then changed the experiment setting to “with probability 99%, choose one from ‘a’ or ‘b’ and with probability 1%, choose one from ‘a’,’b’,’c’,or ‘d’.” Here is the result:



It occurs that in this case, the OpenMP implementation of KMP has good speedups against a sequential implementation. We tried to run KMP and Rabin-Karp over an alphabet of size 2, but keep the rest of the setup the same as Case 1 (uniformly random short patterns). It turns out that in this case, the speedup of KMP over Rabin-Karp is still around 1.1x. The performance of KMP is significantly better if we enlarge the alphabet. If we think further, an alphabet size of 2 means it is pretty hard to predict whether we will have a match and whether we will need to apply the failure function, while a larger alphabet size means most of the time, we won't even be able to match one letter of the pattern. The unpredictability of KMP means it is hard to predict the memory access and computational patterns of KMP when we have ½ chance of matching one more character for the pattern and ½ chance that we need to apply failure links, and applying failure links will incur additional cycles. Thus, the performance of KMP will still greatly depend on whether we mostly have matches or more likely we don't match, especially on modern processors with branch prediction and prefetching. Theoretical analysis, on the other hand, suggests the asymptotic runtime of KMP does not depend on alphabet size or expected length of matches at a given position. Thus, we conclude that the features of the patterns would still incur very different workload for KMP, which is why we are seeing (1) poor performance of KMP over a small alphabet and random patterns and (2) poor speedup of our OpenMP implementation of KMP.

To confirm our conjecture, we use "perf" tool to measure the branch misses of sequential implementation of KMP and Rabin-Karp in the test cases here:

Setup	Branches	Branch Misses	Miss Rate
One > 99%, KMP	15,674,906	1,631,753,802	0.96%
One > 90%, KMP	31,494,217	1,532,401,543	2.06%
Two > 99%, KMP	112,461,822	1,285,289,871	8.75%
One > 99%, RK	12,413,533	486,875,986	2.55%
One > 90%, RK	13,173,010	486,806,155	2.71%
Two > 99%, RK	12,415,327	486,862,65	2.55%

Here, “Two > 99%” refers to our last test case where with 99% probability, we randomly sample from ‘a’ and ‘b’. Then we run the “two > 99%” test, except that in one scenario, we force the patterns to start with ‘a’ (so we are still likely to match first letter of the pattern) and in the another scenario, the patterns start with ‘c’ (so we are unlikely to match first letter of the pattern).

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%

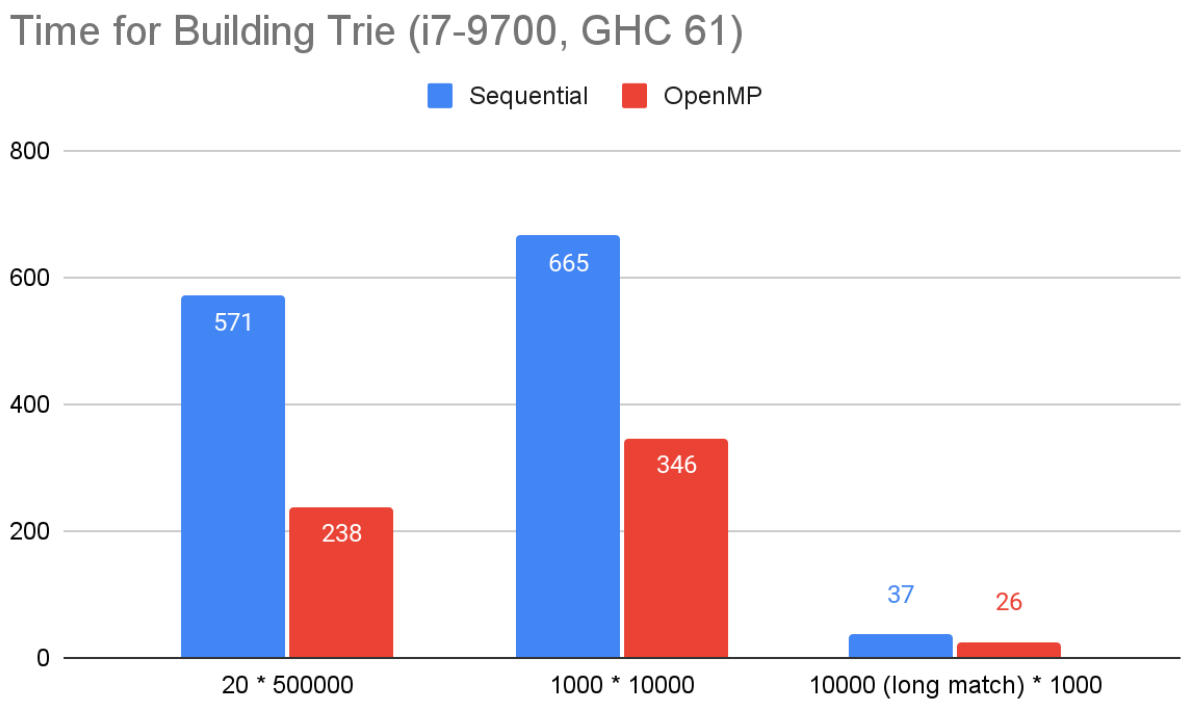
As we can expect, if the pattern strings are forced to start with ‘a’ and we do not make further interventions, then at each step, we have 50% chance that the text will match the next character of the pattern and 50% chance we will have to apply the failure function. This yields a very high branch miss rate. On the other hand, if pattern strings are forced to start with ‘c’, since the text is mostly ‘a’ and ‘b’s, at each step, we are more likely to realize that the text does not match the next character of the pattern, so we end up with a much lower branch miss rate.

These results give us the painful conclusion: even though the theoretical analysis of KMP does not say it will perform bad when we are expected to have long match streaks or if at each step we have 50/50 chance of matching the next character of pattern, modern CPU architecture means lack of predictability will be bad.

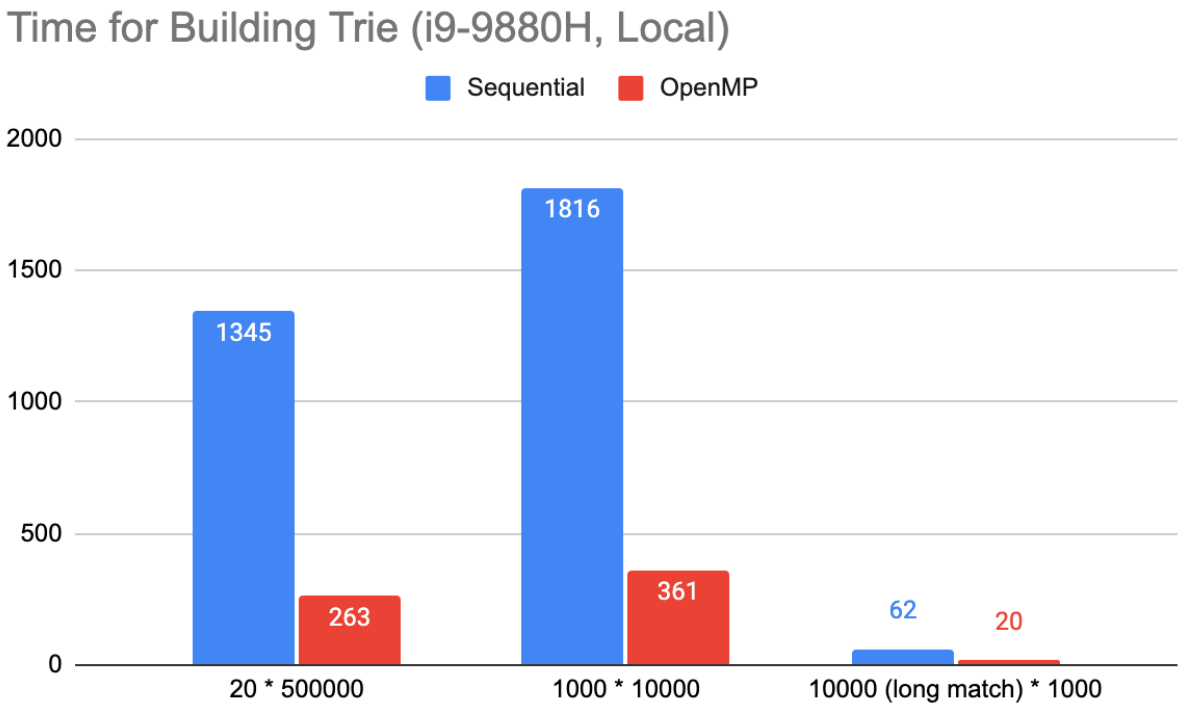
Case 5: Huge Amount of Patterns

In the final test case, we will test the performance of algorithms against a huge amount of patterns. Since we are going to match text against huge amounts of patterns, it is only meaningful to measure the performance of implementations based on Aho-Corasick. In the tests, we are going to generate a pattern size of 10MB. The rest of the setup is similar to Case 1 (uniform patterns) and Case 3 (patterns share a long prefix of repeated characters): the alphabet has size 4, the patterns have the same lengths, the text has size 8MB. We tested against uniformly random patterns of lengths of 20 (500,000 patterns) and 1000 (10,000 patterns), as well as 1000 patterns where there are 9900 letter ‘a’s at the beginning and a random suffix of length 100.

Here is a comparison of construction time based on the sequential implementation of Aho-Corasick (units are “milliseconds”):

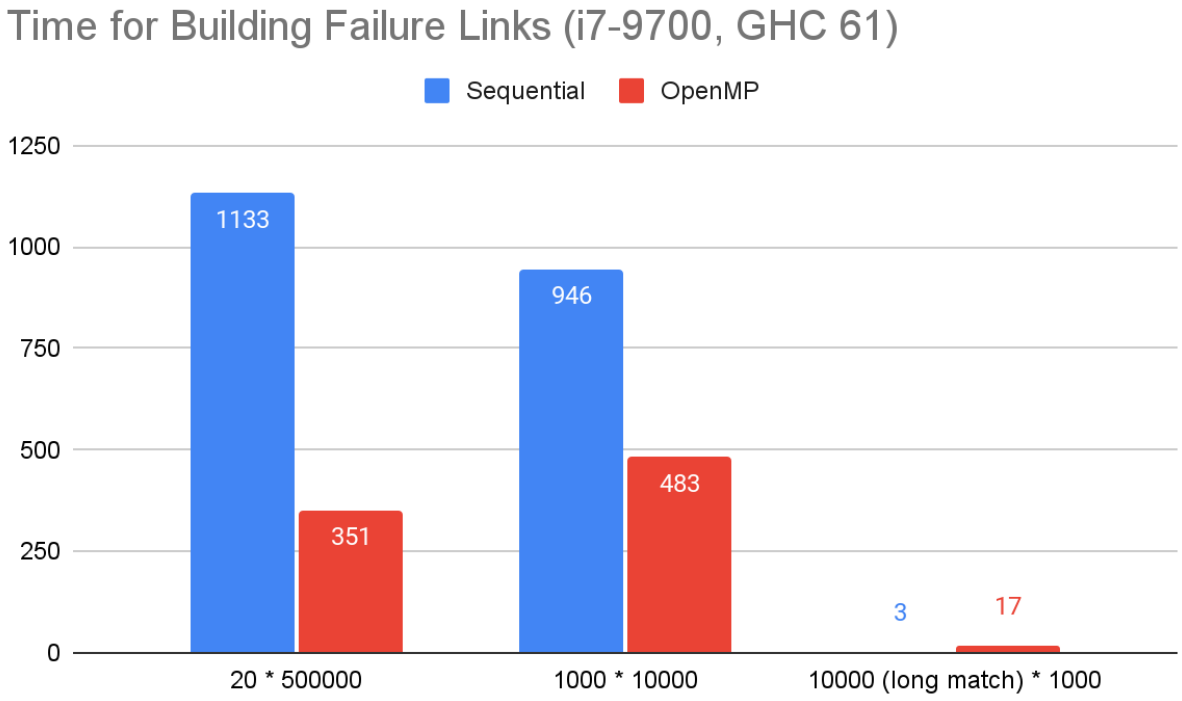


For those uniformly randomly generated patterns, the OpenMP implementation for parallel insertion can achieve a 2.4x speedup over the serial implementation when the patterns have length 20 and 1.9x speedup over the serial implementation when the patterns have length 1000. Notice that since patterns are sampled uniformly randomly from all possible strings over the given length, when we walk down the Trie, at first few levels we are likely to encounter nodes that are already there. When we go further down, we likely need to create a lot of new nodes, and creating new nodes might be an inherently sequential part of the algorithm. Another problem with allocating new nodes is threads might stall when we are waiting for memory allocation and initializing the newly allocated space. The stalling issue becomes evident when we try to measure the number of instructions per cycle: for the 500000 patterns of length 20 case, the Trie building phase only yields 0.37 instructions per cycle. For the 10000 patterns of length 1000 case, the Trie building phase yields 0.29 instructions per cycle, which is even worse. For the “1000 patterns with longest repeated characters in front” case, since we are using __sync_compare_and_swap primitive to atomically test if there is an edge with the given label from a Trie node (and insert new node if there is none), there would be a lot of coherence traffic since most of the time the patterns are going down along the same shared path (i.e. corresponding to the prefix of repeated characters), which is also not extremely efficient. When the tests are run on my personal laptop (Core i9-9880H, 8 cores, 16 threads with hyperthreading), we see the uniformly randomly generated patterns case exhibits a 5x speedup against the sequential implementation and the “1000 patterns with longest repeated characters in front” case exhibit a 3x speedup:

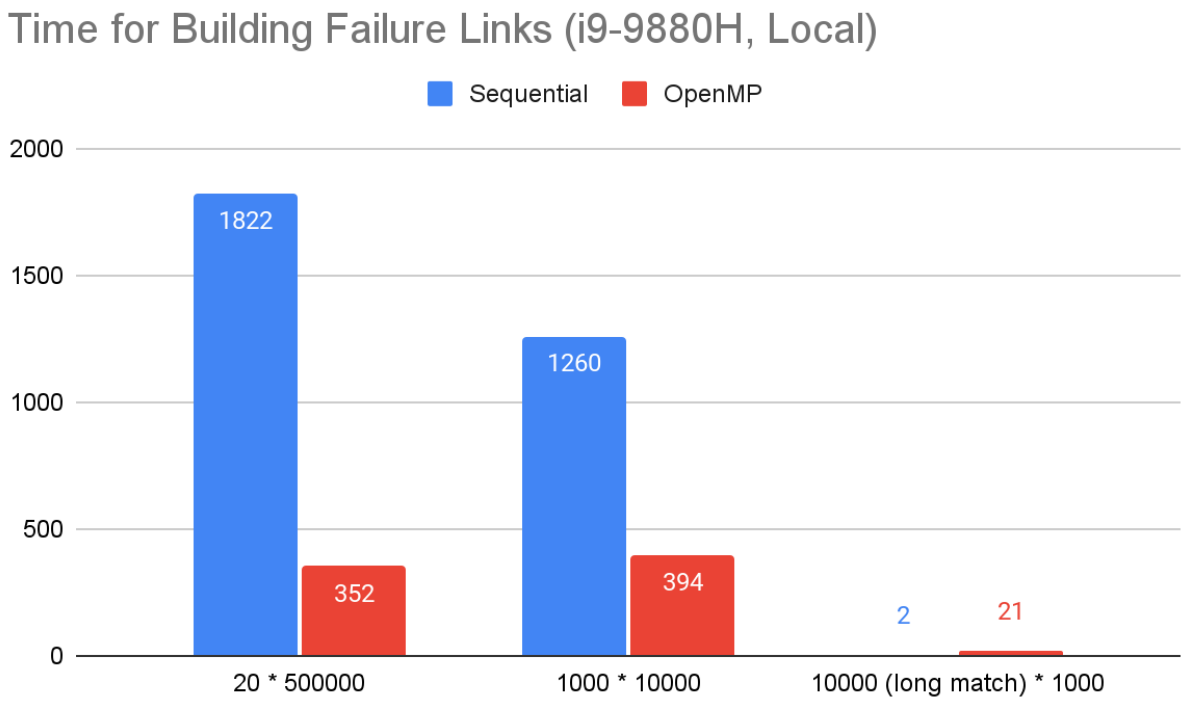


The high performance gain resulting from hyperthreading, as well as a low instruction per cycle count, suggests that we are unable to achieve perfect speedup due to the large amount of cycles wasted on stalling for memory and/or coherence traffic.

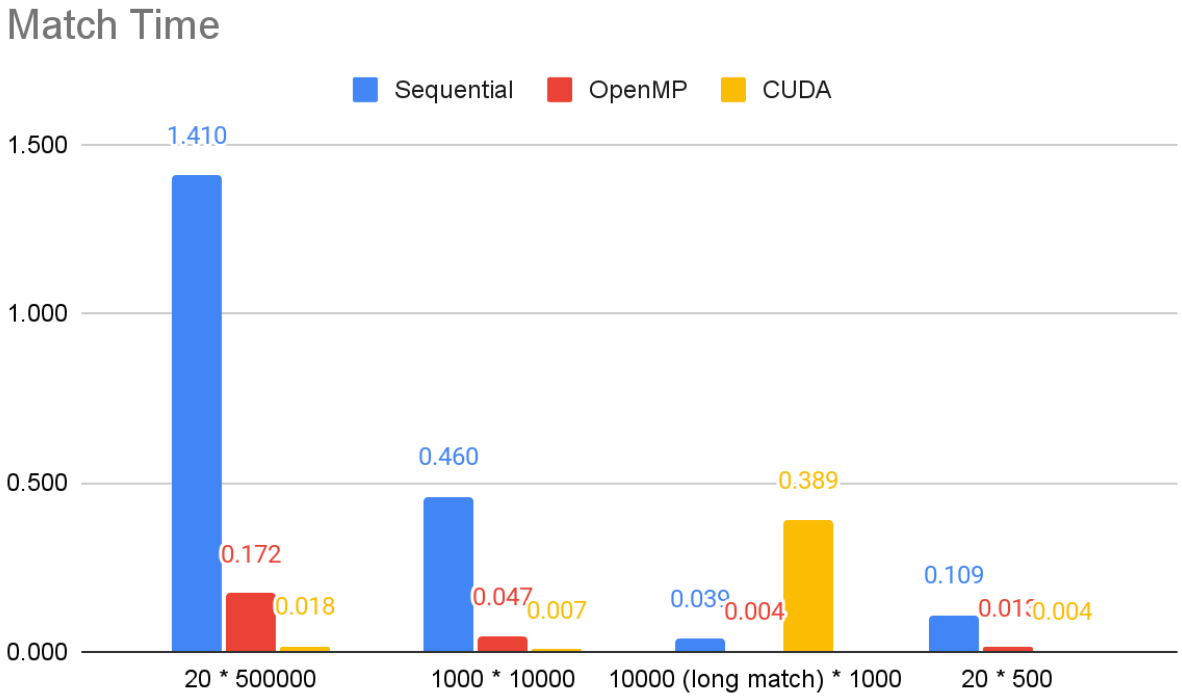
The performance of the “parallel construction of failure link” phase is provided below:



We are able to achieve a 3x speedup over the case where we have 500000 patterns of length 20, and a 1.95x speedup over the case where we have 10000 patterns of length 1000. Notice that our parallel BFS algorithm would make exploring the nodes in the same frontier parallel, so there is synchronization after finishing exploring the same frontier. Furthermore, if there are only a few nodes in the current frontier (like the case where most of the levels only consist of a single node corresponding to the “repeated character”), then the algorithm is almost sequential. The ideal case is if we have few levels but each level contains sufficiently many nodes, which is the case when we have 500000 patterns of length200. This case will better utilize the parallel BFS construction routine. Local tests suggest we can even achieve 5x speedup on this case on a machine with 8-core CPU and hyperthreading:



The time spend on matching is summarized in the below table (unit in seconds):



Theoretical analysis says when the sum of length of patterns and the text length is fixed, we should have approximately the same runtime. The experimental results, on the other hand, greatly deviates from this assumption. Even though the sum of length of patterns is fixed, the case where patterns share a very long common prefix means the Trie has only a few nodes and the Trie can easily fit into the L3 cache (but not yet into L1/L2 cache). For example, if we have 1000 patterns where the patterns all start with 9900 ‘a’s and the rest are random, a rough estimate of nodes in Trie is $9900 + 100 * 1000 = 109900$ since we can reuse the nodes to store the common prefix. On the other hand, for the “uniform pattern” cases, the number of nodes is roughly the same as the sum of lengths of patterns and there is little compression involved. In this case, the automaton does not fit into the L3 cache. This results in a huge difference in cache miss rates. Furthermore, as described before, the memory access pattern for matching along the Aho-Corasick automaton is harder to predict and prefetch since the matching process can be roughly described as a pointer chasing process. Thus, the cache misses resulting from accessing the automaton will translate to a huge runtime penalty. Indeed, if we scale down to 500 patterns of length 20, we see a 10x speedup at OpenMP and sequential implementations of Aho-Corasick. The rest of the performance difference may be explained by the difference in branch misses: when the patterns and the text are built of blocks where blocks consist of a very long prefix of repeated character, then our matching process is highly predictable: we just go down along the Trie. On the other hand, if the patterns and the text are all uniformly randomly generated, then it is hard to predict where we will end up with. When we measure the branch prediction failure rate using perf, we see that the former case has a negligible branch prediction miss rate, while the later case has a miss rate of approximately 7%.

We also see that in our particular case, the OpenMP implementation shows perfect speedup, if not yet superlinear speedup, against the sequential implementation. As we have

argued, in this case the size of the Aho-Corasick automaton does not fit into L1/L2 cache, so the runtime is not bound by arithmetic computations but instead by communication caused by irregular memory access patterns and cache misses. When we are using OpenMP, if a core experiences a cache miss when it tries to access some node of the automaton, it might be able to check if the node exists in the cache of some other cores (using say, BusRd commands). If so, it might be able to fetch the node from other core's L1 and L2 cache instead of going to memory or the slower L3 cache. This can partly explain why we have a perfect speedup when we have so many patterns that the automaton won't fit into the L1 cache or L2 cache.

Finally, as expected, when we expect we have long matches from a given start position, CUDA implementation of Aho-Corasick does not work well. In particular, if there are 500,000 patterns, then we are expected to match for a longer time than if there are only 500 or 10,000 patterns. This explains the subtle difference in the runtime of the CUDA implementation of Aho-Corasick.

Benchmarks on Real World Data

We ran following experiments on the GHC61 machine. It has an 8 core Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz. It has NVIDIA GeForce RTX 2080 B GPU which supports CUDA compute capability 7.5 [8]

We benchmarked our implementation on 2 real world datasets -

1. DNA Corpus - Human genome information consisting of characters a, t, g, c [10].
2. Protein Corpus - This is a set of 4 files, which were used in the famous DCC 1999 article "Protein is incompressible" [9]. The alphabet size is 26 (lower case english alphabets).

We look at two kinds of problems -

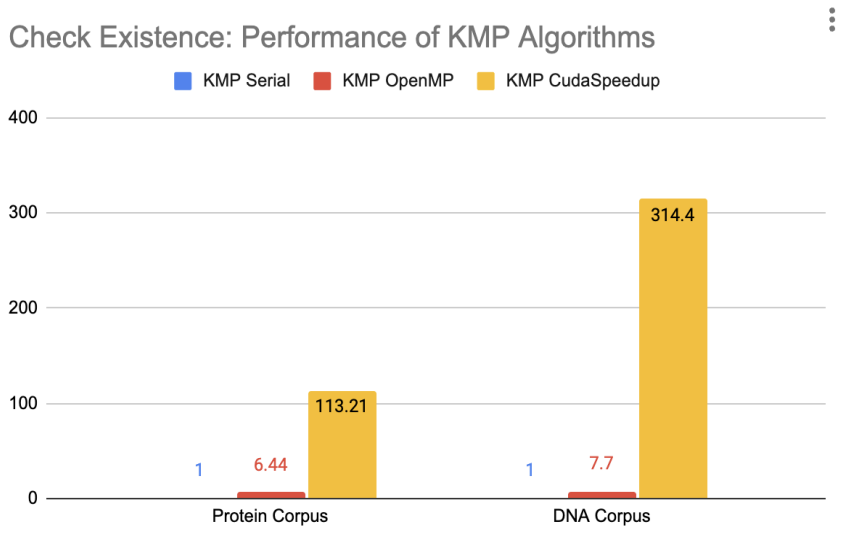
1. **Count Occurrences:** Given a set of patterns, count the total number of occurrences of patterns in the text (summed over count of all patterns). This has application in Bio-Informatics
2. **Check Existence:** Given a set of patterns, check if there exists at least one pattern in the text. Network Intrusion Detection Systems find malicious patterns in packets using the Check Existence Problem

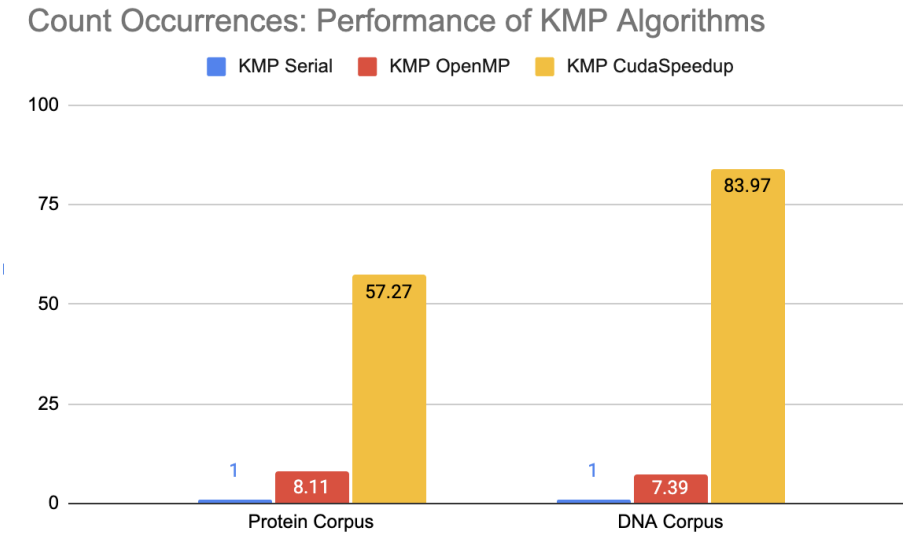
For the "Count Existence" problem, we generated patterns by first randomly sampling substrings of lengths 4, 16, 64, 256, 1024, 2048 from the input texts. Then with some probability, we will shuffle the excerpts to create new patterns, which may or may not have matches. Then we randomly sampled some of those patterns to create a final list of patterns. We have a roughly equal number of patterns with the given length. In the end, we created a list of 260 patterns for the DNA corpus and another list of 260 patterns for the protein corpus.

Likewise, for the "Check Existence" problem, we generated patterns by randomly sampling substrings of lengths 256, 1024, 2048 from text. Then with some probability, we will create new patterns by randomly shuffling the substrings. Initially, we used smaller pattern lengths too for this problem, however smaller pattern lengths might mean that some algorithm just ends up lucky: it might be able to find a match early so it can quit, hence the results will not

be interpretable. Therefore, we opted for only larger pattern lengths for the “Check Existence” problem. We created a list of 204 patterns for the “check existence” problem over the DNA corpus and a list of 234 patterns for the “check existence” problem over the protein corpus.

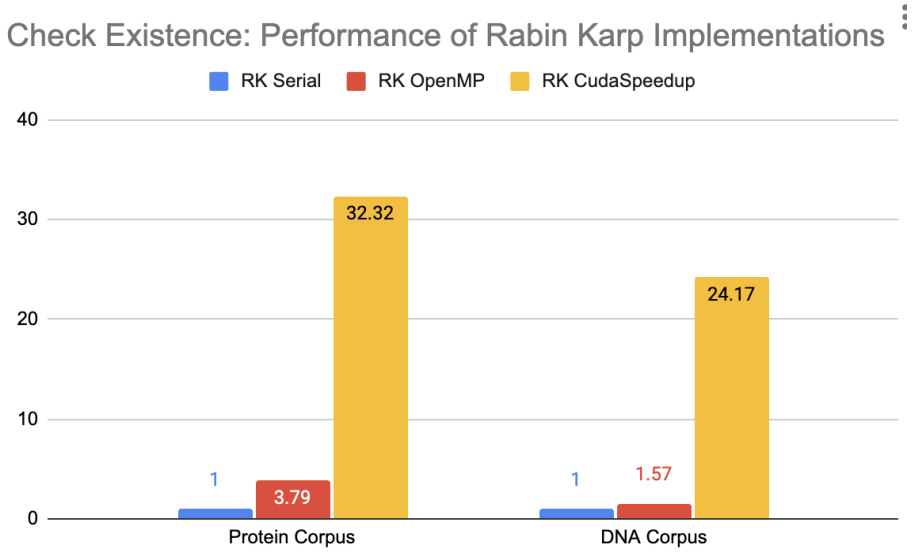
Note: In all of the following graphs, we don’t compare performances across corpuses. Each speedup number is relative to the performance of the algorithm on the respective corpus.

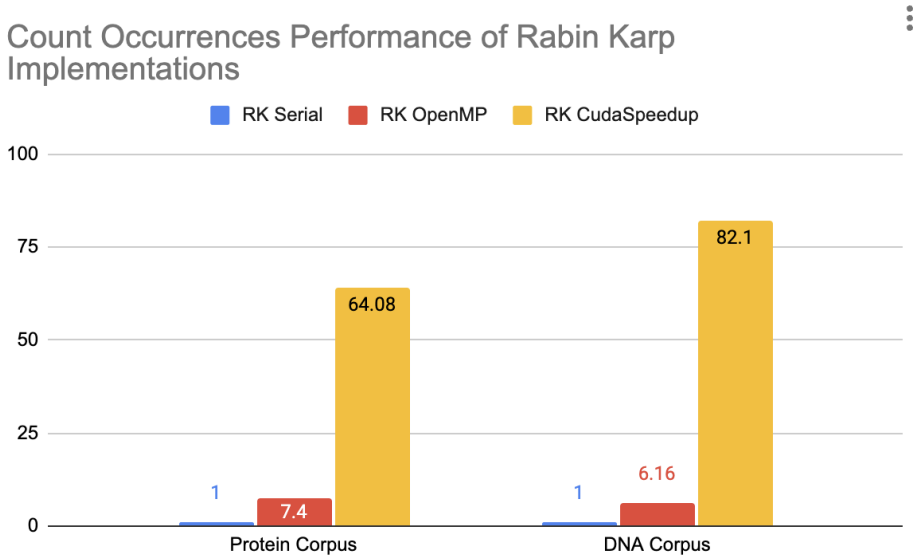




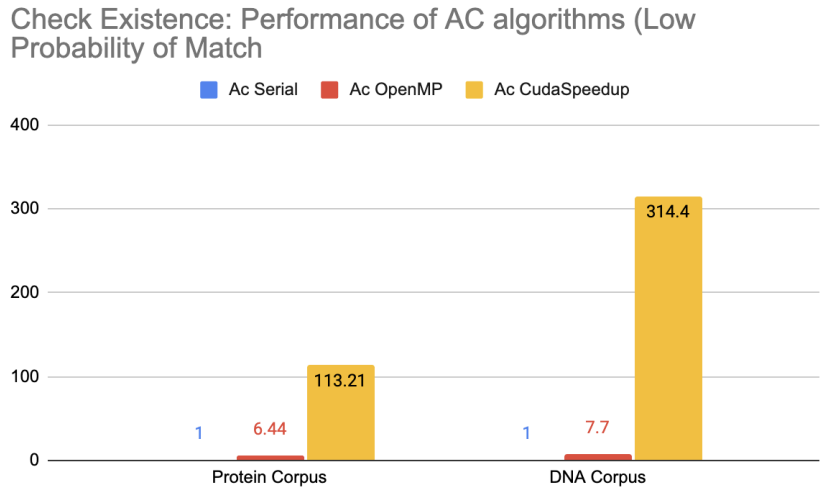
The above graph says that KMP OpenMP on protein corpus is 8.11x faster than KMP serial on Protein Corpus. Likewise, KMP Cuda on DNA corpus is 83.97x faster than KMP serial on DNA corpus.

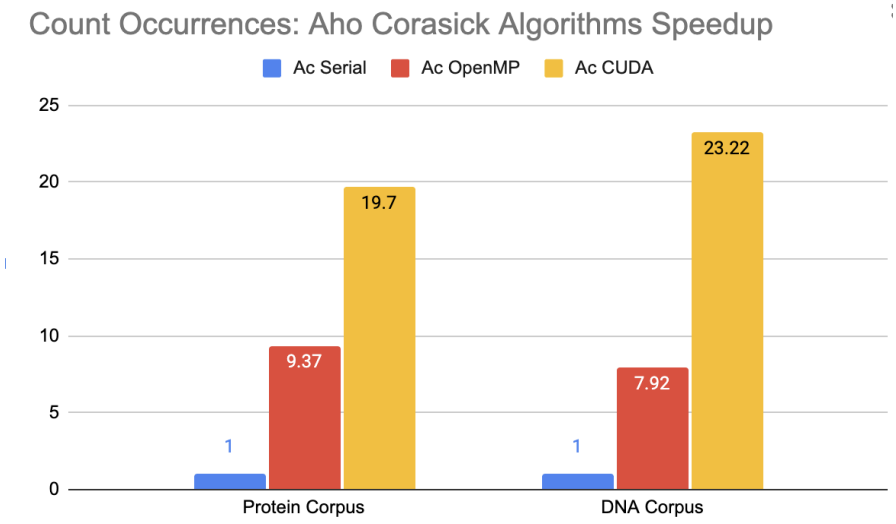
Above graphs show that CUDA implementation of KMP algorithm achieves significant speedup compared to the serial implementation of KMP. OpenMP implementation also achieves almost expected speedup. We observed good performance on both corpuses for both kinds of problems.



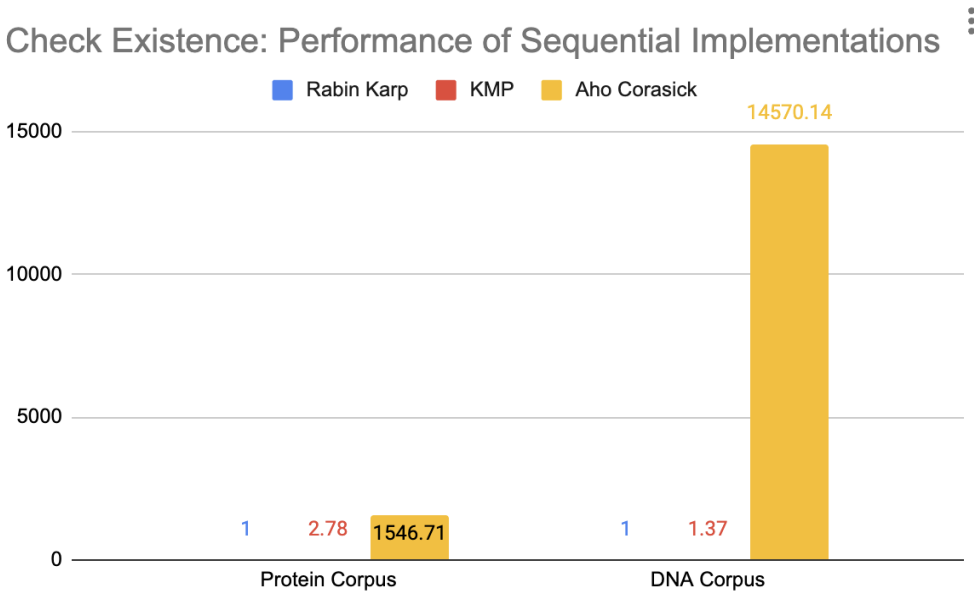


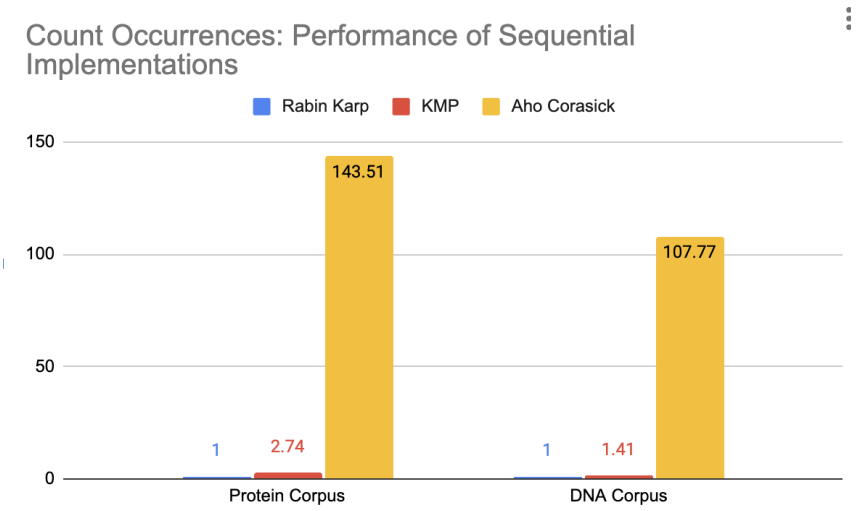
For Rabin Karp as well, the CUDA and OpenMP implementations achieve good speedup compared to the sequential implementation of Rabin Karp.





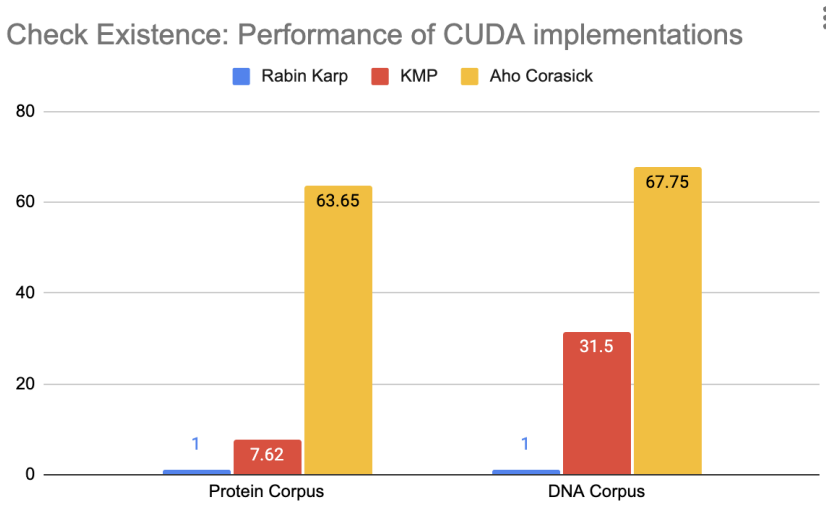
We see good speedup in parallel implementations of Aho-Corasick as well. Performance of both Rabin Karp and Aho-Corasick is affected by the alphabet size. The alphabet size of protein corpus is 26 while for DNA corpus is 4. Hence, we see that they do better on the DNA corpus as compared to Protein Corpus.

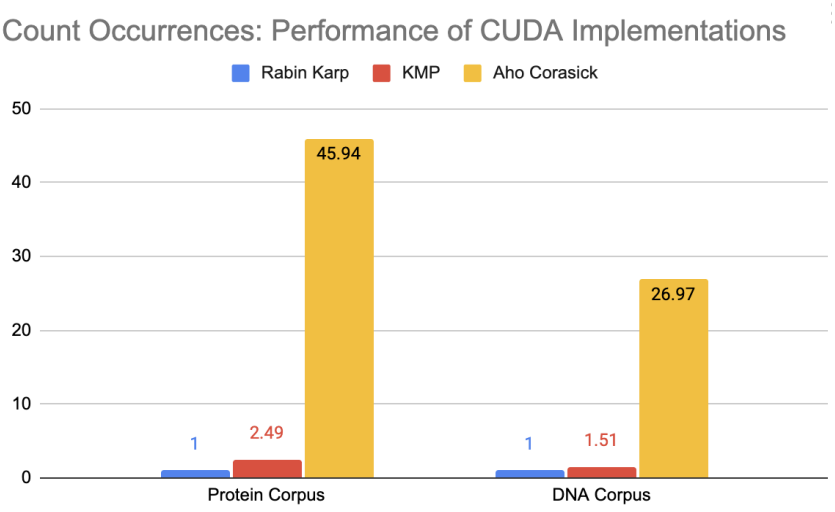




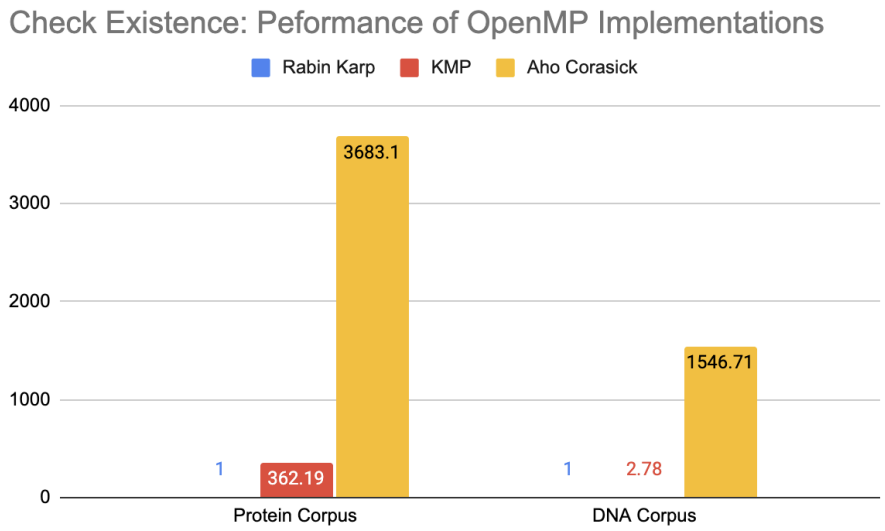
The above plot says that serial implementation of of KMP on protein corpus is 2.74x times faster than the serial implementation of Rabin Karp on protein corpus. Likewise, serial implementation of Aho-Corasick on DNA corpus is 107.77x faster than the serial implementation of Rabin Karp on DNA corpus.

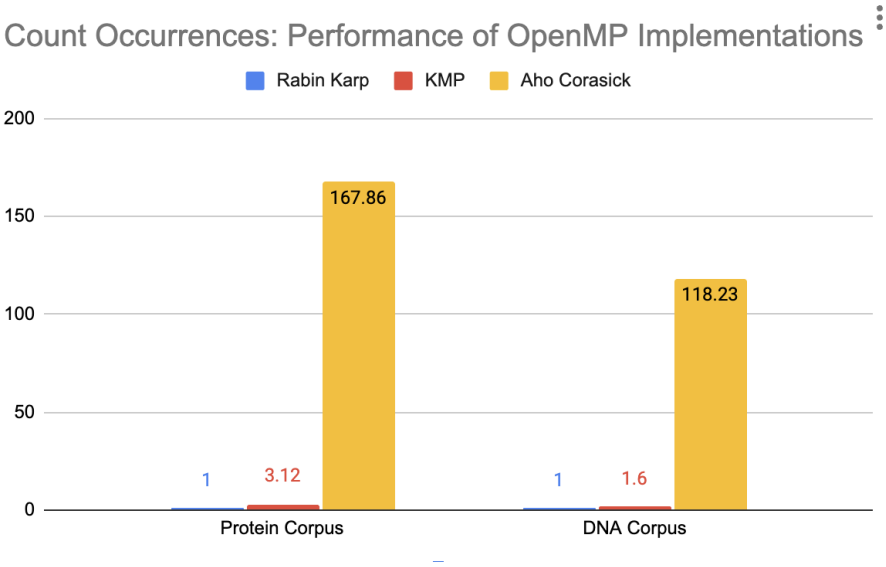
Above plots compare sequential implementations of three algorithms. We see that Rabin Karp is slowest while KMP is almost 2x faster than Rabin Karp. Aho-Corasick does really well and confirms our theoretical understanding of Aho-Corasick's performance in case of multiple patterns.





Above graphs show, that CUDA implementation of KMP is around 2x faster than CUDA implementation of Rabin Karp for count occurrences problem while Aho-Corasick being the best.





We obtain similar results for OpenMP implementation.

One thing that we observe here is that the observed speedup is less for DNA corpus than Protein Corpus in both CUDA and OpenMP implementations of AC. This can be explained by the fact that on an average, the DNA Corpus has more matches (722K) as compared to the Protein Corpus (241K). If there are tons of long matches, then all algorithms essentially start working like naive string search. Again, we should note that the “Check Existence” problem does not require us to go over the entire text: once a thread is able to find a good position where there is a match, the entire matching process will stop. Thus, it is natural to see some “superlinear” or “sublinear” speedups.

Conclusion

Our experimental results suggest that Rabin-Karp has relatively stable, though poor, performance. This arises from the fact that Rabin-Karp has regular memory access patterns. For each position of the text, Rabin-Karp does a constant amount of work, except if there are matches and we want to use brute-force to verify hashes do not give us false matches. Rabin-Karp is slow because it uses expensive operations like multiplication and taking modulus, which are costly on modern CPU architectures. Over a single pattern, KMP performs better than Rabin-Karp in general, though KMP achieves greater speedup when branching predictors can predict whether we are going to match against the pattern further or if we are not going to match against the pattern further. KMP does not perform well if at each point, we have 50% chance of having a match and 50% chance of not having a match. Even though Aho-Corasick is theoretically equivalent to KMP over a single pattern, the less compact representation leads to a higher cache miss rate than KMP, so the performance is slightly worse.

Over multiple patterns, the sequential and OpenMP implementations of Aho-Corasick have huge advantages over sequential and OpenMP implementations of Rabin-Karp and KMP. This is because Aho-Corasick does not need to iterate through the text for multiple times when there are multiple patterns. Our experimental results suggest that in general, implementing Aho-Corasick on CUDA using the strategy where each thread is responsible for one possible starting position of the match leads to best results, though this strategy seems to suffer when for most positions, we are able to match the pattern for extended length.

Finally, our experiments on parallel construction of Aho-Corasick automata using OpenMP show that lock-free insertion algorithms to Trie and parallel BFS can often lead to some, though not perfect, speedup. When the size of the patterns does not fit into L2 cache or L3 cache, though, the matching phase would slow down significantly due to cache misses and high cost of communication.

References

[1] Wikipedia. Knuth-Morris-Pratt Algorithm.

https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

[2] Wikipedia. Rabin-Karp Algorithm.

https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm

[3] Wikipedia. Aho-Corasick Algorithm.

https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm

[4] Pfaffe, Philip, et al. "Parallel string matching." European Conference on Parallel Processing. Springer, Cham, 2016.

[5] Lin, Cheng-Hung, et al. "Accelerating string matching using multi-threaded algorithm on GPU." 2010 IEEE Global Telecommunications Conference GLOBECOM 2010. IEEE, 2010.

[6] Park, Neungsoo, Soeun Park, and Myungho Lee. "High performance parallel KMP algorithm on a heterogeneous architecture." Cluster Computing 23.3 (2020): 2205-2217.

[7] Thambawita, D. R. V. L. B., Roshan G. Ragel, and Dhammike Elkaduwe. "An optimized Parallel Failure-less Aho-Corasick algorithm for DNA sequence matching." 2016 IEEE International Conference on Information and Automation for Sustainability (ICIAfS). IEEE, 2016.

[8] Wikipedia. GeForce 20 Series. https://en.wikipedia.org/wiki/GeForce_20_series

[9] Nevill-Manning, Craig G., and Ian H. Witten. "Protein is incompressible." Proceedings DCC'99 Data Compression Conference (Cat. No. PR00096). IEEE, 1999.

[10] Giovanni Manzini. A corpus for DNA compression algorithms.

https://people.unipmn.it/manzini/dnacorpus/

Distribution of Credit

We believe that we both contributed roughly equally to this project.