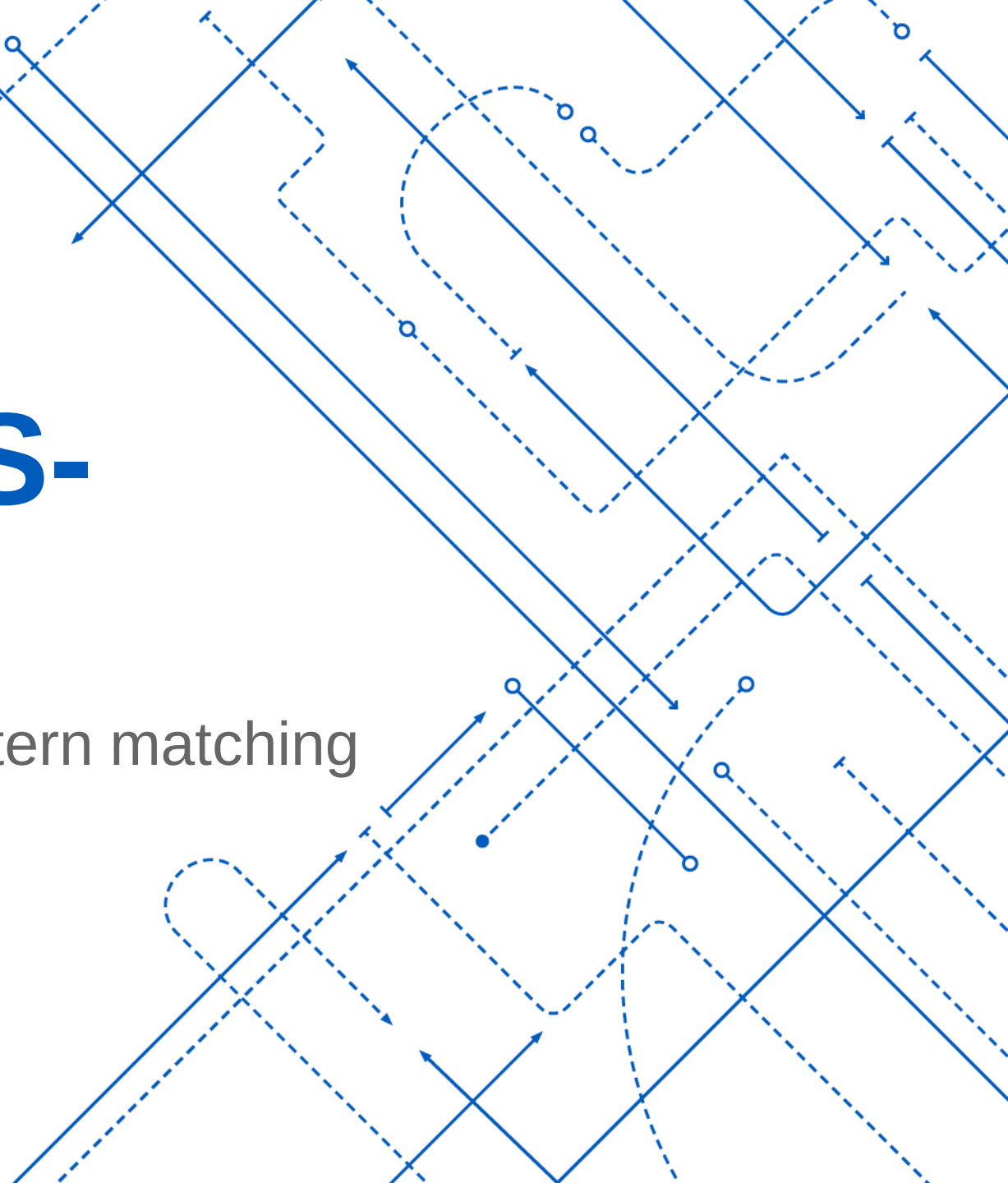
KNUTH-MORRIS-PRATT

Exploring the KMP algorithm for pattern matching

Priya Rao (UB IT Name: prao4)

CSE 702 SEM

University at Buffalo The State University of New York

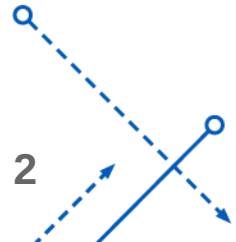


Problem Statement

There are several applications to exact string matching algorithms

1. Plagiarism detection
2. DNA Sequencing
3. Forensics
4. Spelling Check
5. Search Engines

Not all algorithms fare well when it comes to matching patterns against an extremely large string. Especially when implementing in sequential. In parallel as well, a naïve algorithm would not provide the best results.



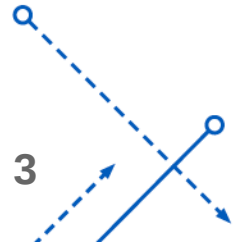
Knuth Morris Pratt

WHY KMP?

KMP algorithm provides an advantage - It is guaranteed to be worst-case efficient.

As observed for the sequential algorithm:

The preprocessing time is always O(m) and the searching time is always O(n)



How does KMP work?

- There are two main aspects to KMP:
  1. **Pre-processing** : This involves parsing through the pattern alone (O(M) time and space) – an array of prefix-suffix match is created
  2. **Searching** : This involves parsing through the string using the pre-processed array and the pattern array (O(N) time)

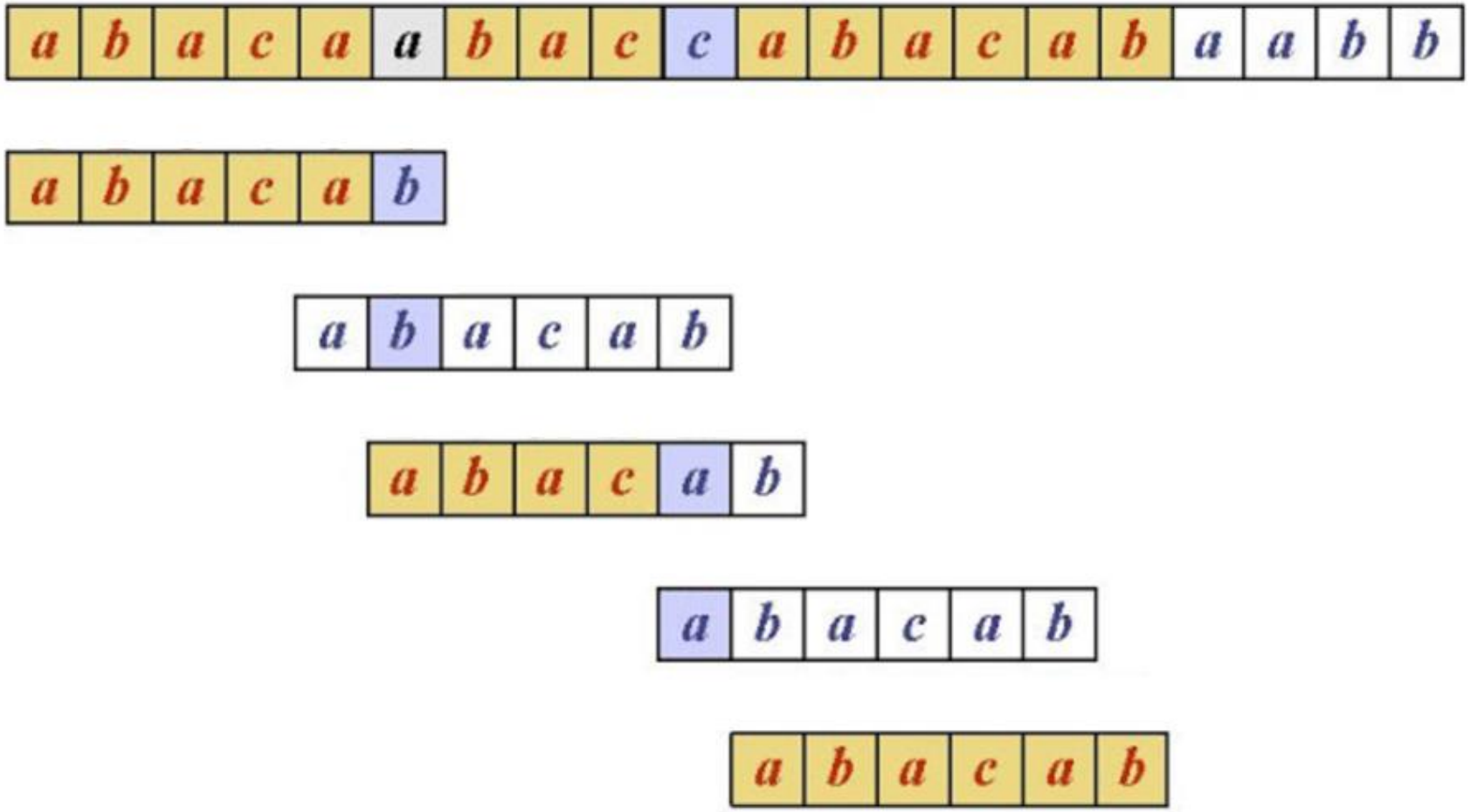
Pre-processing

The length of the longest proper prefix in the (sub)pattern that matches a proper suffix in the same (sub)pattern.
Example for cell with index 5. Sub-pattern is "abacab"
Proper Prefix: a, ab, aba, abac, abaca
Proper Suffix: b, ab, cab, acab, bacab

Let us find out the common string in proper prefix and proper suffix. We get "ab".
Length of longest one (here the only one common) is 2.
Therefore, value of cell with index 5 is 2.

x	0	1	2	3	4	5
P[x]	*a*	*b*	*a*	*c*	*a*	*b*
f(x)	0	0	1	0	1	2

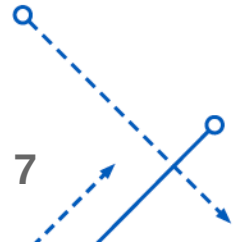
Pattern Searching



Parallel Approach

The approach for the parallel implementation of KMP is as follows:

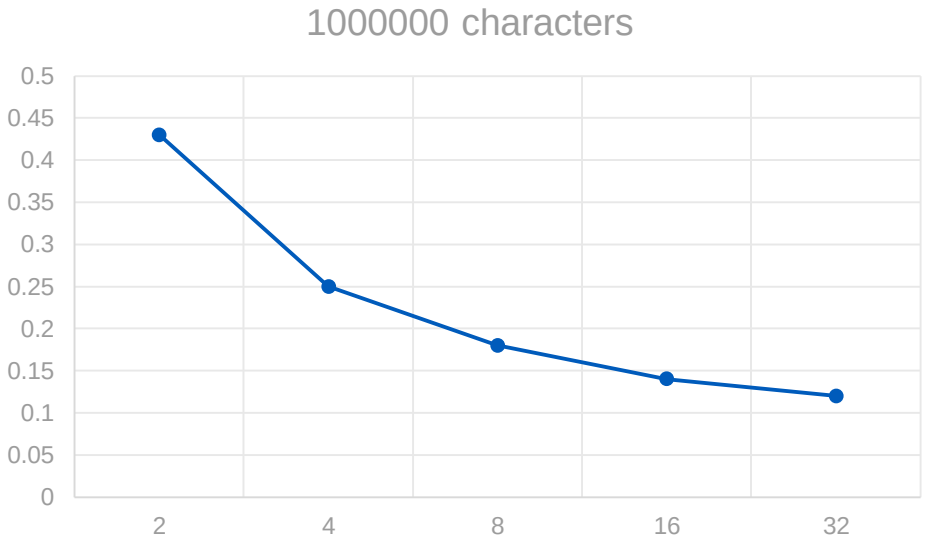
1. Split the main string (of length S) into sub-strings per thread such that each thread gets (S/p) length of string where p is the number of threads.
2. The pattern table would be available in the shared memory from which all threads would derive.
3. Apply the serial KMP Search algorithm in each of the threads, keeping a track of the matches so far by updating the global counter as and when a match is secured.
4. The final results are tabulated in the master thread.



Results

- For target string of 100,000 characters, the results were obtained as follows:

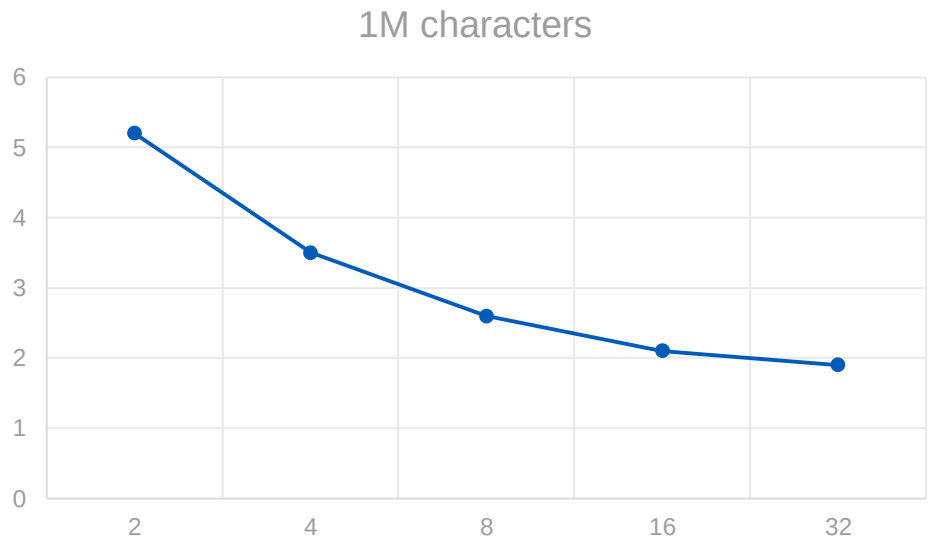
Threads	Time (secs)
2	0.43
4	0.25
8	0.18
16	0.14
32	0.12



Results

- For target string of 1,000,000 characters, the results were obtained as follows:

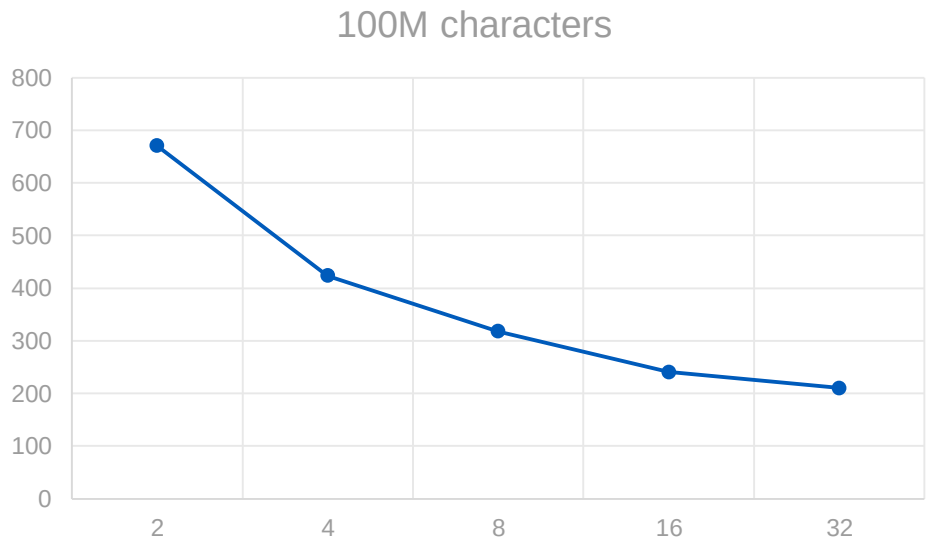
Threads	Time (secs)
2	5.2
4	3.5
8	2.6
16	2.1
32	1.9



Results

- For target string of 100,000,000 characters, the results were obtained as follows:

Threads	Time (secs)
2	670.2
4	423.4
8	317.7
16	240.5
32	210



Conclusions

- Parallelization through threads resulted in a significant speed up
- A good part of the code is serial itself. So theoretical speed up can’t be achieved as one would expect
- Factors possibly affecting performance – shared memory, scheduling, compiler optimization, synchronization, etc

References

- https://en.wikipedia.org/wiki/OpenMP
- https://cse.buffalo.edu/~vipin/nsf/docs/Tutorials/OpenMP/omp-II-handout.pdf
- https://www.openmp.org/resources/tutorials-articles/
- https://www3.nd.edu/~zxu2/acms60212-40212/Lec-12-OpenMP.pdf
- https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
- https://ieeexplore.ieee.org/document/6234095

Thank You