KNUTH-MORRIS-PRATT

Exploring the KMP algorithm for pattern matching

Priya Rao (UB IT Name: prao4)

University at Buffalo The State University of New York

Problem Statement

String matching applications are all around us – ranging from DNA pattern matching to plagiarism detection.

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

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

As observed for the sequential algorithm:

The preprocessing time is always O(m)

The searching time is always O(n)

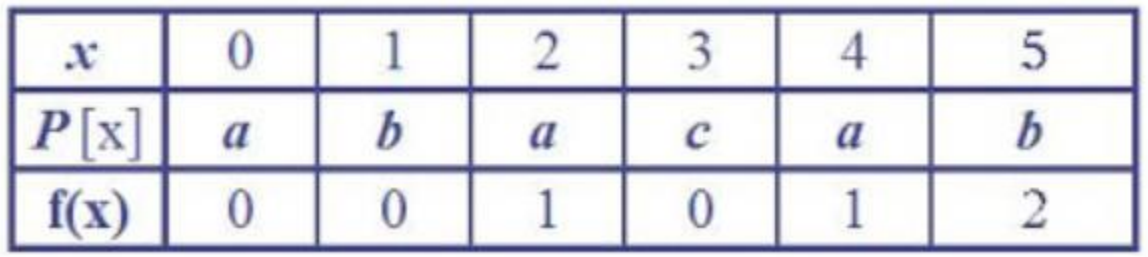
How does KMP work?

- 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

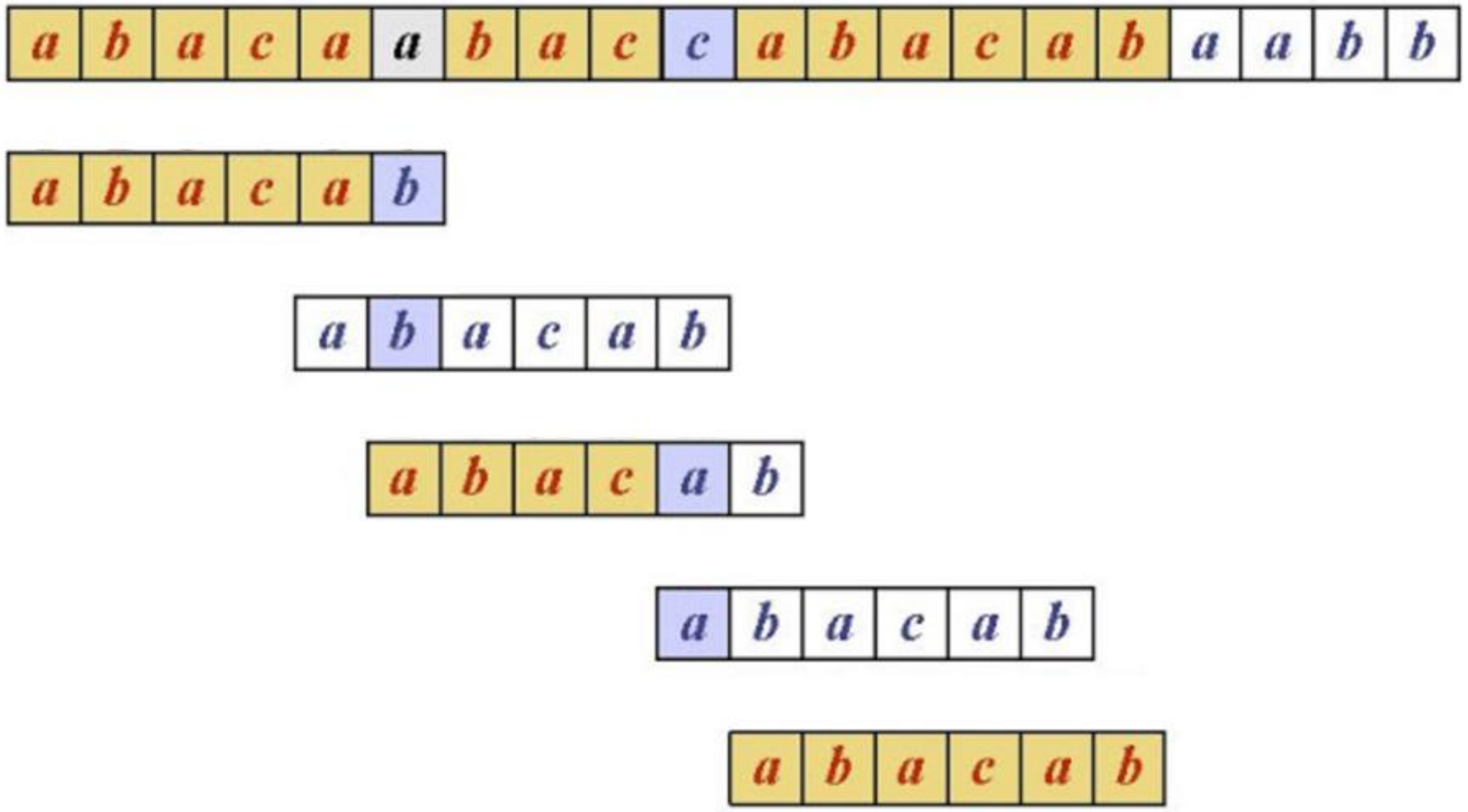
The length of the longest proper prefix in the (sub)pattern that matches a proper suffix in the same (sub)pattern.
Example for cell with index 5. Sub-pattern is "abacab"
Proper Prefix: a, ab, aba, abac, abaca
Proper Suffix: b, ab, cab, acab, bacab

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:

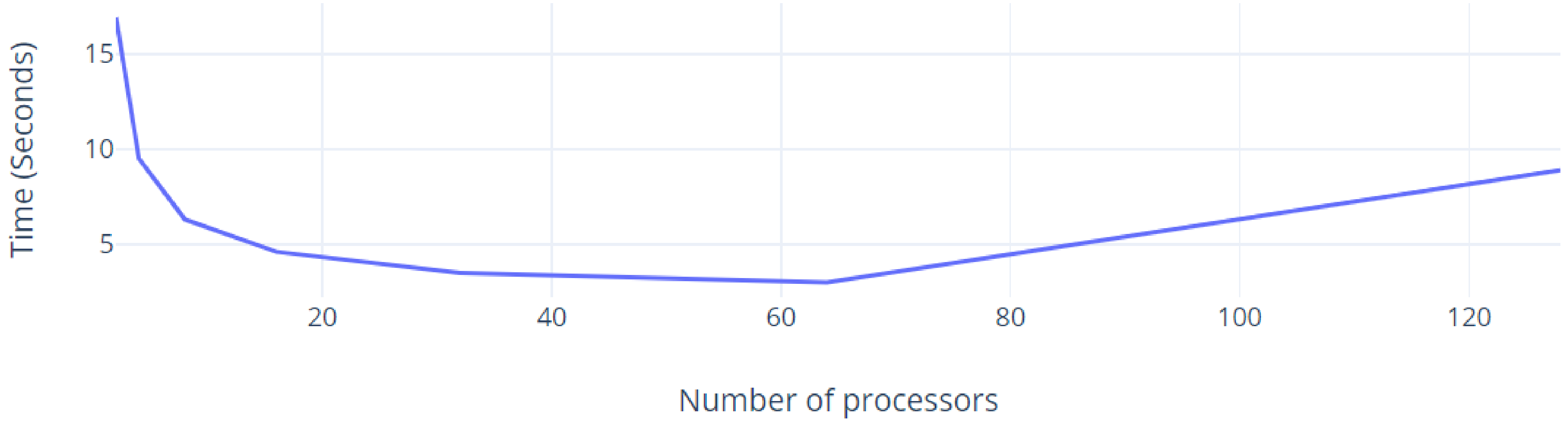
1. Split the main string (of length S) into sub-strings per processor such that each processor gets (S/N) length of string where N is the number of processors
2. Apply the serial KMP Search algorithm in each of the processors, keeping a track of the "perfect" matches so far
3. Concatenate the results by doing a prefix sum. Nth processor would contain the final count.

Parallel Approach

4. In case of a partial match in processor 1, this processor would send the length of the partial pattern to the next processor (processor 2).
5. If processor 2 has received this partial length, then it checks for pattern[length – partial length] at the beginning of the string it has.
6. If this is found as a match, then increment the count and the start index of the matched pattern would be at current index – pattern length.

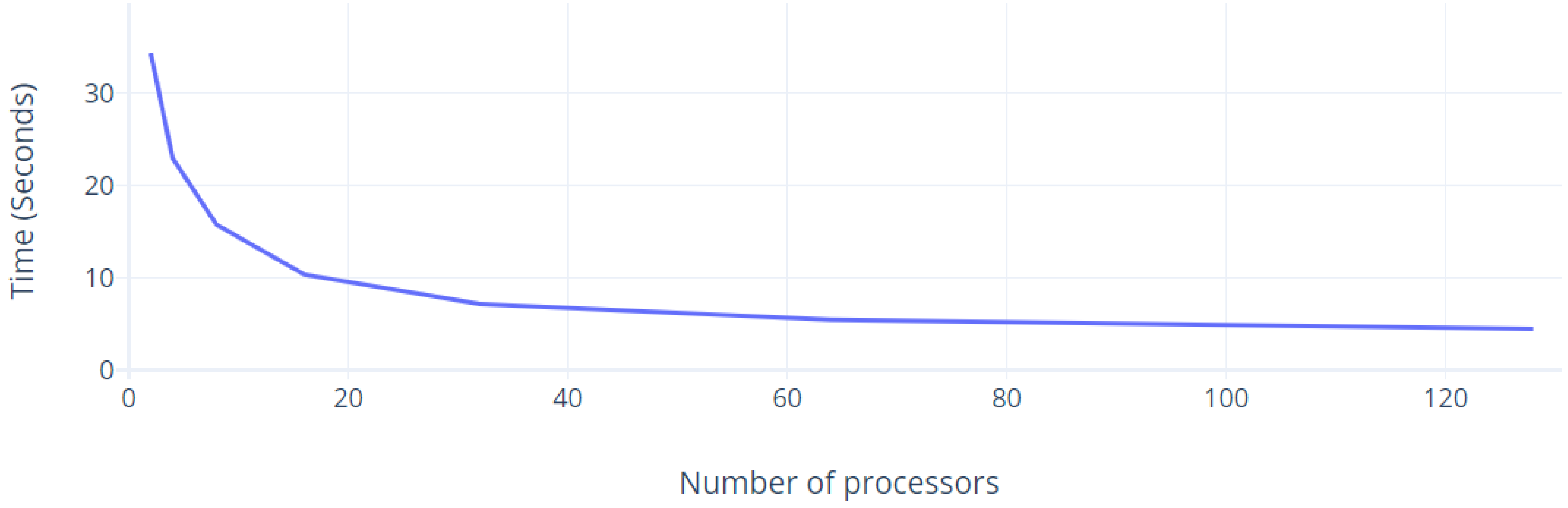
Results (S = 1 M, P = 100)

Processors	Seconds
2	16.9
4	9.5
8	6.3
16	4.6
32	3.5
64	3.0
128	8.9



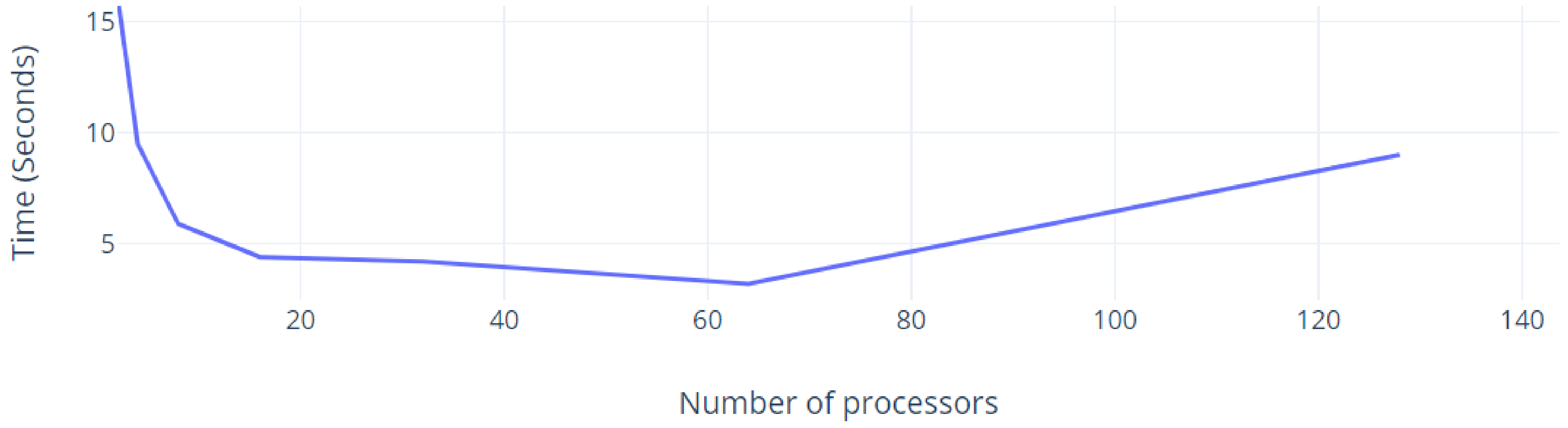
Results (S = 100 M, P = 100)

Processors	Seconds
2	34.3
4	22.9
8	15.7
16	10.3
32	7.1
64	5.4
128	4.4



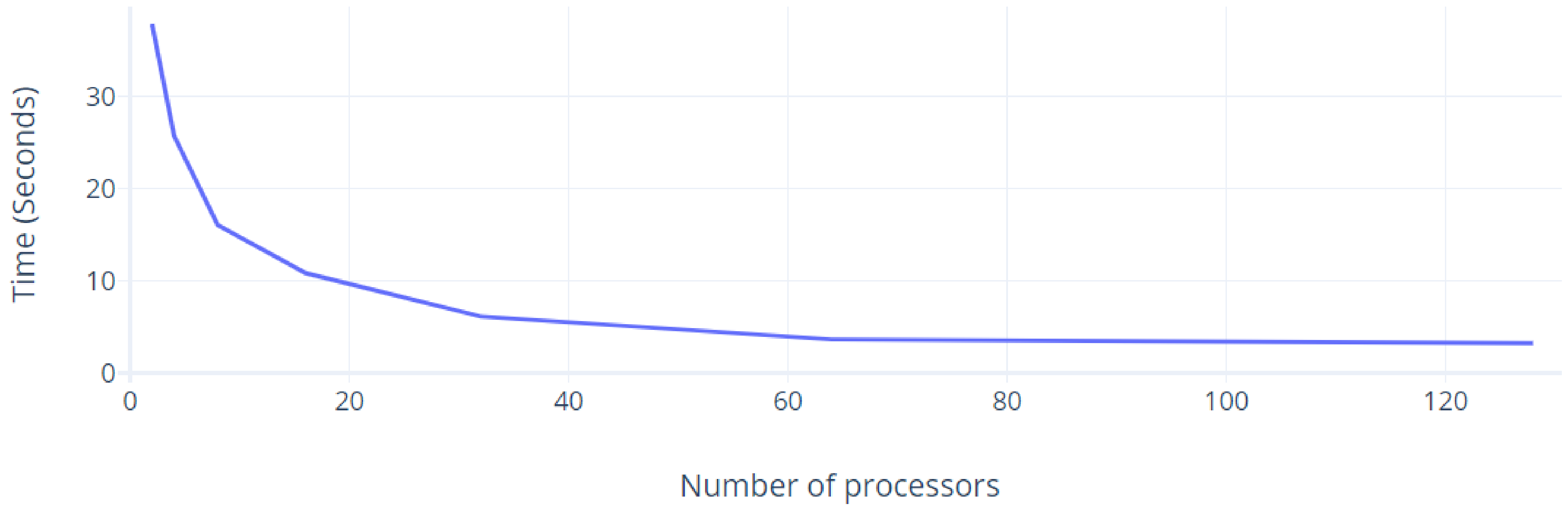
Results (S = 1 M, P = 10000)

Processors	Seconds
2	16.24
4	9.5
8	5.9
16	4.4
32	4.2
64	3.2
128	9.0



Results (S = 100 M, P = 10000)

Processors	Seconds
2	37.8
4	25.7
8	16.0
16	10.8
32	6.1
64	3.6
128	3.2



Observations

- Parallelization resulted in a significant speed up of the algorithm.
- Parallelization performed almost similarly for any length of the pattern and any number of occurrences of the pattern in the string
- After a certain point though increasing the number of processes causes a communication overhead which results in increased/stagnation of time

University at Buffalo The State University of New York

Challenges

- No documentation to address scenario which handles a pattern itself being so large that it does not fit into a single processor
- Queueing time for large number of processors
- Generating large strings for this purpose

References

https://www.inf.hs-flensburg.de/lang/algorithmen/pattern/kmpen.htm

https://web.stanford.edu/class/cs97si/10-string-algorithms.pdf

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.258.8768&rep=rep1&type=pdf

https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

https://curc.readthedocs.io/en/latest/programming/MPI-C.html

https://ieeexplore.ieee.org/document/6618720

Thank You