KMP PARALLEL ALGORITHM FOR PATTERN MATCHING (OPEN MP)

UBIT: rbammidi

Professor: Dr. Russ Miller

University at Buffalo The State University of New York

Content

- Need of pattern matching
- KMP Algorithm
- Parallel KMP
- Failed parallelization attempts
- Debugging with slurm
- Results

Need of pattern matching

- Pattern matching is used to determine whether source files of high-level languages are syntactically correct.
- Many fingerprint recognition methods are in use to perform fingerprint matching out of which pattern matching approaches is widely used.
- Pattern matching enables users to find the locations of particular DNA subsequences in a database or DNA sequence.
- Searching for word in the large log files dump
- Validating the information received from the client before writing into DB.

Knuth-Morris-Pratt (KMP)

The Knuth-Morris-Pratt (KMP) algorithm is an algorithm that is used to search for a pattern in a given text in $O(m + n)$ time (where m and n are the lengths of pattern and text).

KMP Algorithm ~ 2 Step Process

Step 1: Pre-process the Pattern

step 1.1: Build the LPS table from the pattern

Step 2: Iterate through the Text and Pattern and check for the existence of the pattern in the text

Components and Terminology of KMP Algorithm

In the KMP algorithm, we have two terms, proper prefix and suffix

A proper prefix of the pattern will be a subset of the pattern using only the beginning portion (the first index), or the first few indices of the pattern except the last character

Pattern : a b c d a b c

- a
- a b
- a b c
- a b c d
- a b c d a b c

Components and Terminology of KMP Algorithm

A proper suffix of any pattern would be a subset of the pattern with elements taken only from the right end of the pattern as in, any number of elements, starting from the last character. Taking the first character of the string is not allowed

Pattern : a b c d a b c

- c
- b c
- a b c
- d a b c

Longest prefix that is also a suffix (LPS)

	A	A	B	A	A	A	A	B	A	A	B
LPS[ ]	0	1	0	1	2	2	2	3	4	5	3

LPS[i] represents longest prefix that is also a suffix till i

Takes O(m) time to generate the LPS array

```
int *kmptable(char *pattern, int len)
{
    int k = 0;
    int i = 1;
    int *table = (int *)malloc(len * sizeof(int));
    table[0] = 0;
    while (i < len)
    {
        if (pattern[k] == pattern[i])
        {
            k += 1;
            table[i] = k;
            i++;
        }
        else if (k > 0)
        {
            k = table[k - 1];
        }
        else
        {
            table[i] = 0;
            i++;
        }
    }

    return table;
}
```

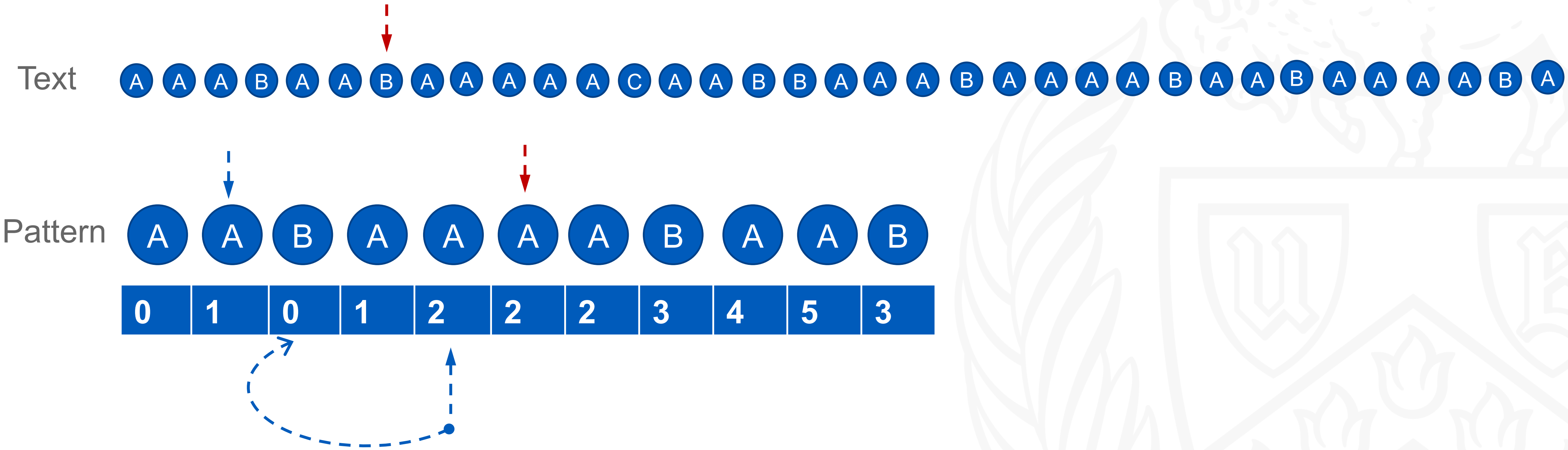
KMP Pattern Matching

Text: A A A B A A B A A A A A C A A B B A A A B A A A A B A A B A A A A B A

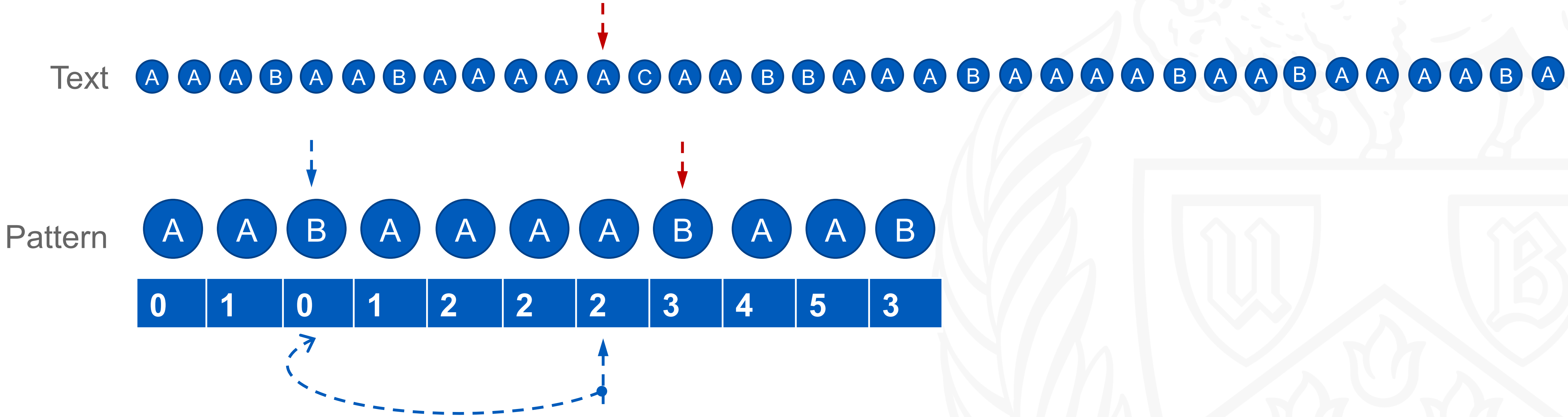
Pattern: A A B A A A A B A A B

A	A	B	A	A	A	A	B	A	A	B
0	1	0	1	2	2	2	3	4	5	3

KMP Pattern Matching



KMP Pattern Matching



KMP Pattern Matching

Text A A A B A A B A A A A A C A A B B A A A B A A A A B A A B A A A A B A

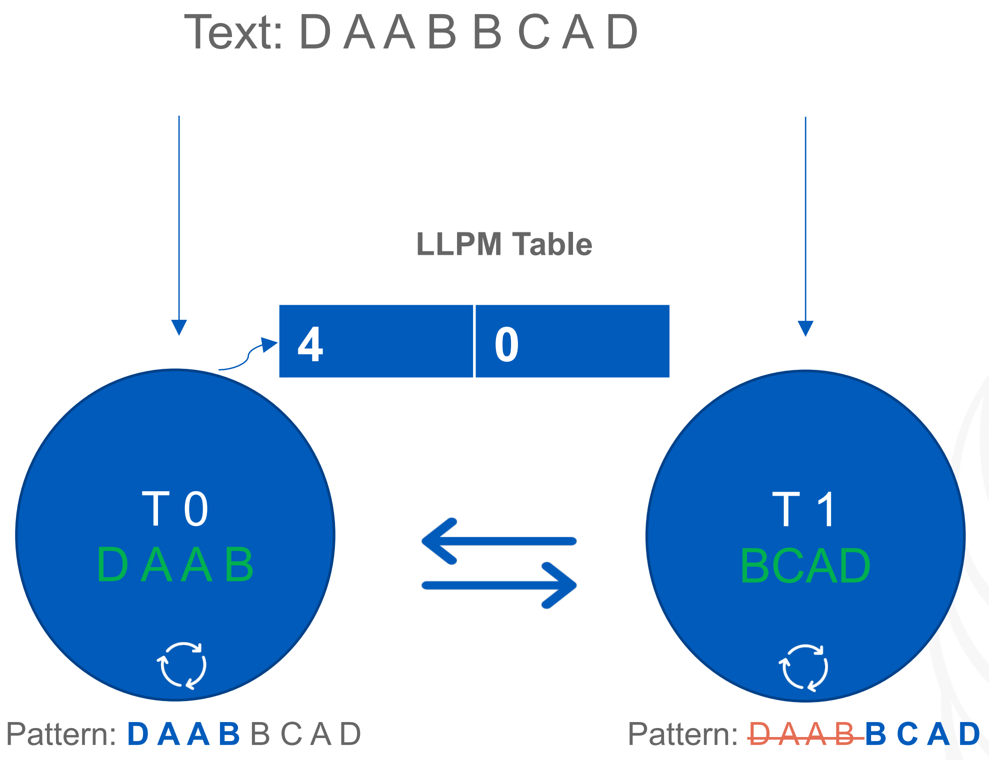
Pattern A A B A A A A B A A B

Parallel KMP Algorithm

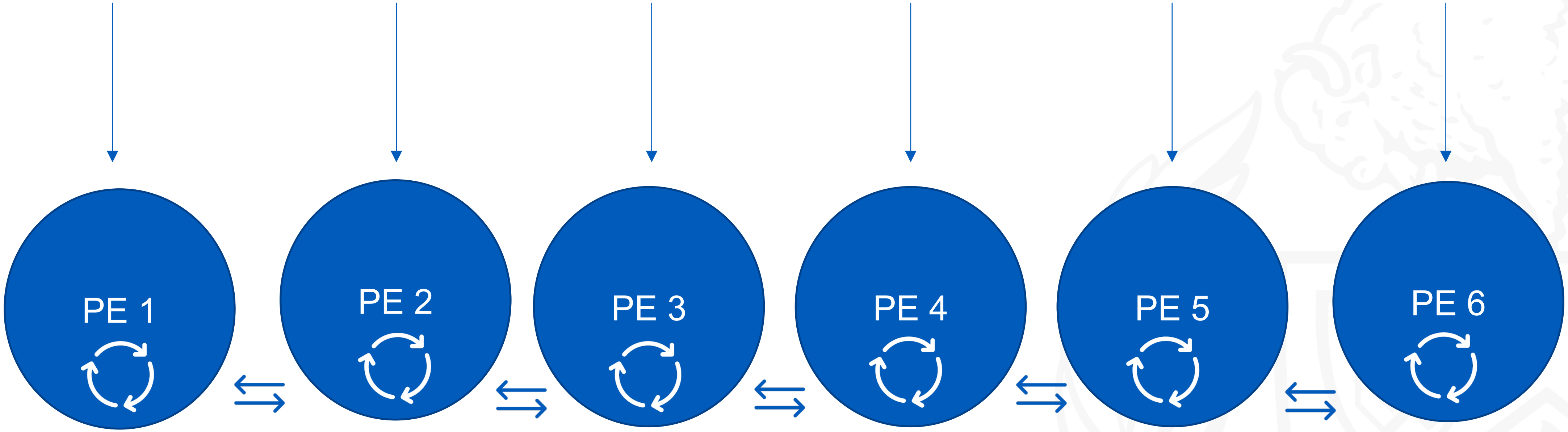
Components and Terminology of Parallel KMP Algorithm

- Shared LLPM Table: llp[i] stores the length of the longest pattern that matches with the text till len(string-1) in the ith thread.
- Cumulative LLPM Table: It holds the cumulative LLPM table information from the processor 0 to processor i
- Non-cumulative LLPM Table: It holds the non-cumulative LLPM table information, which means it doesn't contain the LLPM information from processor 0 (partial LLPM table)

LLPM Table usage



Initial Attempt



Drawbacks:

Every i^{th} processor has to wait until it receives the parallel KMP table from $[0, i-1]$ processors

Parallel KMP Steps (MPI)

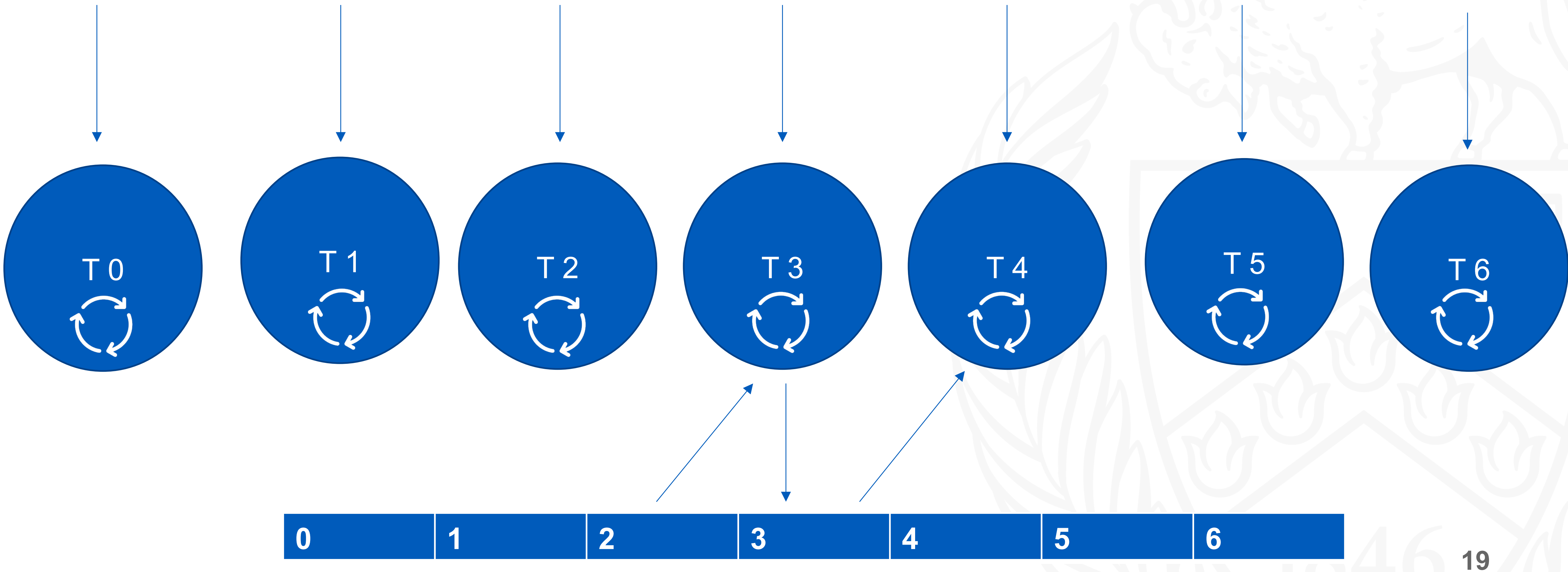
- Split the given text equally of size $(\frac{n}{k})$ $each$ to all the processors ~ Broadcasting
- Each processor executes sequential KMP independently on the given text & pattern
- Every processor checks if the cumulative KMP table is available to receive from its predecessor
- If the cumulative KMP table is not available in the buffer, it receives the non-cumulative KMP table from its preceding processor.
- This process continues till it finds the pattern in the given text.

Parallel KMP Steps (Open MP)

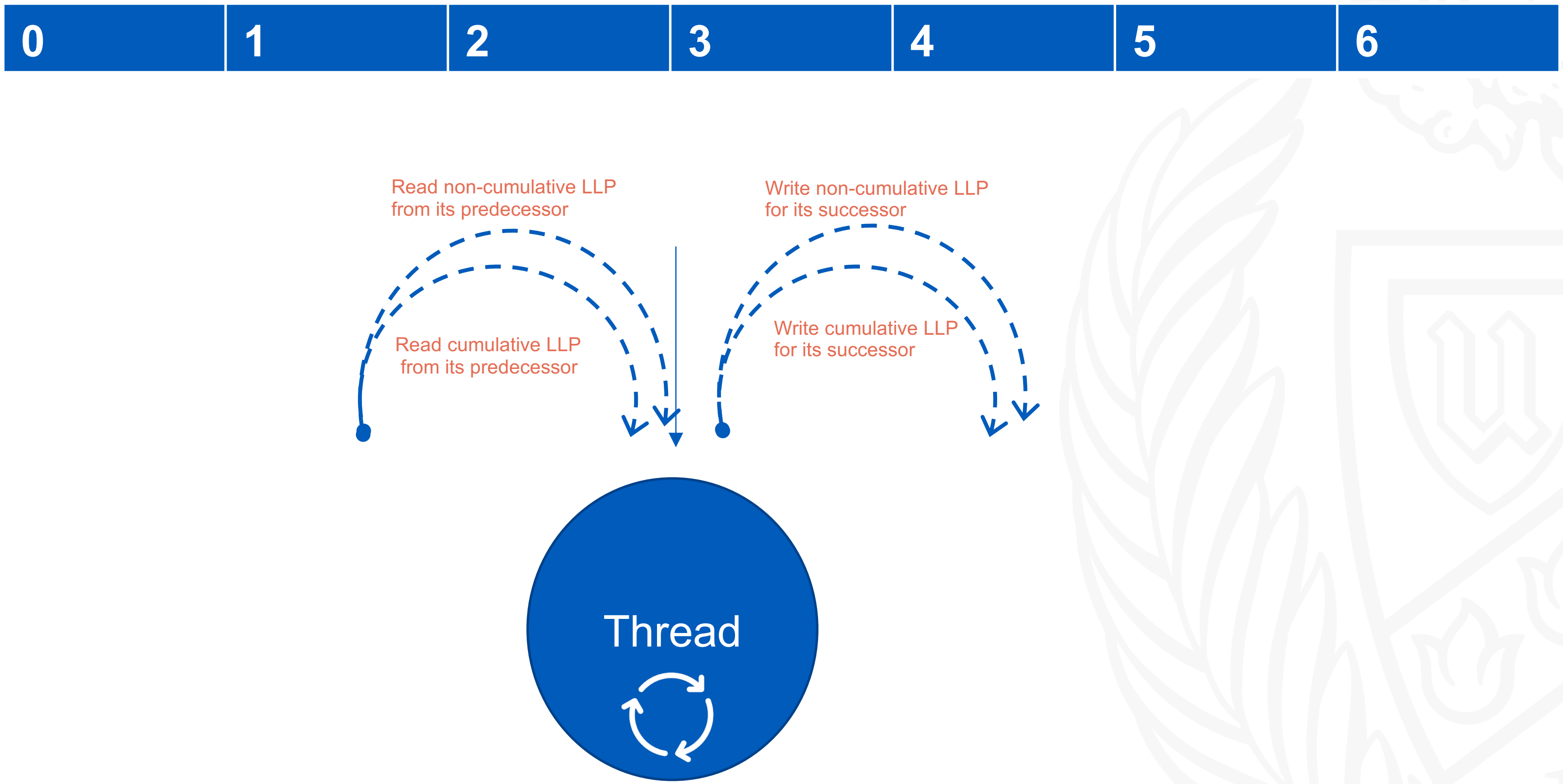
- Split the given text equally of size $(\frac{n}{k})$ $each$ to all the threads
- Each thread executes sequential KMP independently on the given text & pattern
- Every thread checks for the length of largest pattern (say LLP) match at the last index from its predecessor.
- Every thread recompute the LLP in its local scope based on the previous thread LLP.
- This process continues till it finds the pattern in the given text.

Parallel KMP Visualization Using OPEN MP

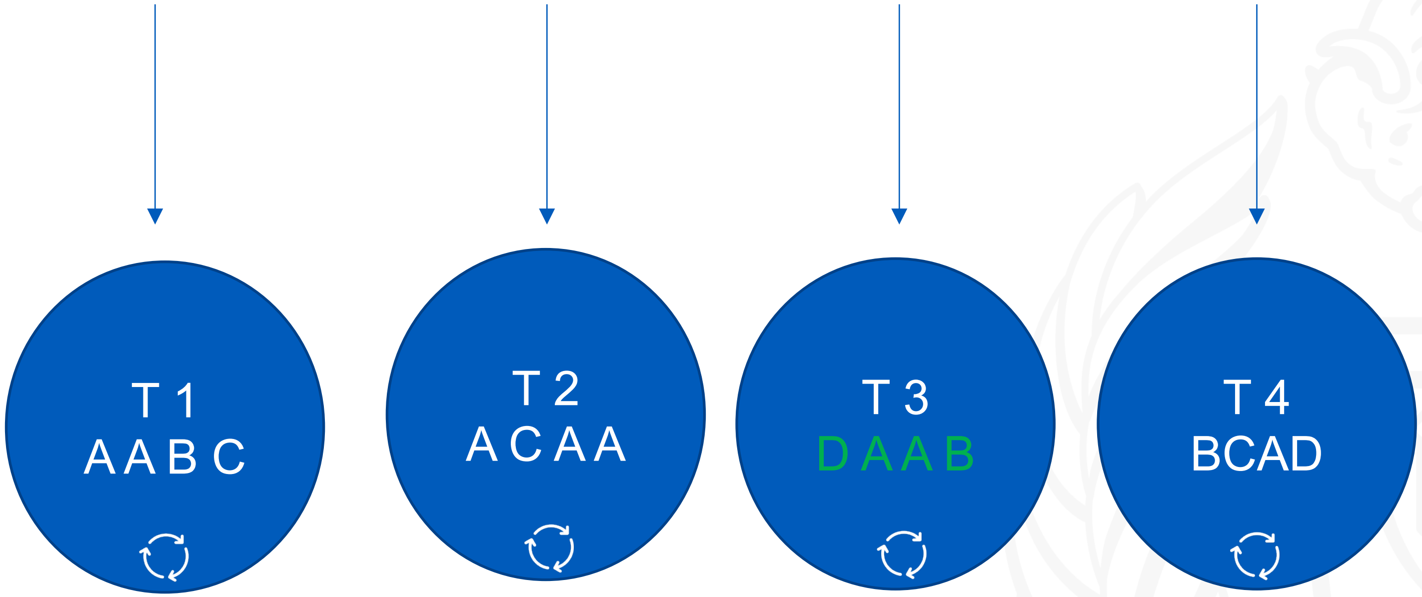
Step1: Process Text & Pattern
Step 2: Write into shared lps table
Step 3: Read from shared lps table, repeat step 1.



Profiling a typical thread

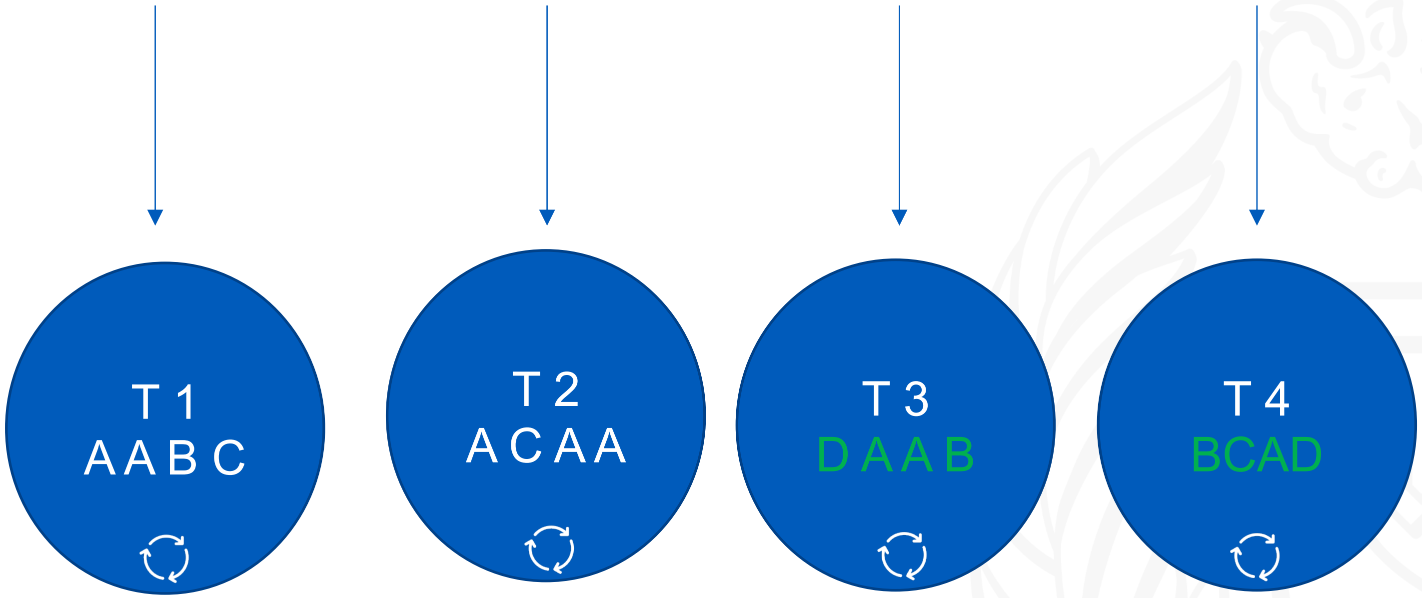


Pattern existence in a single processor



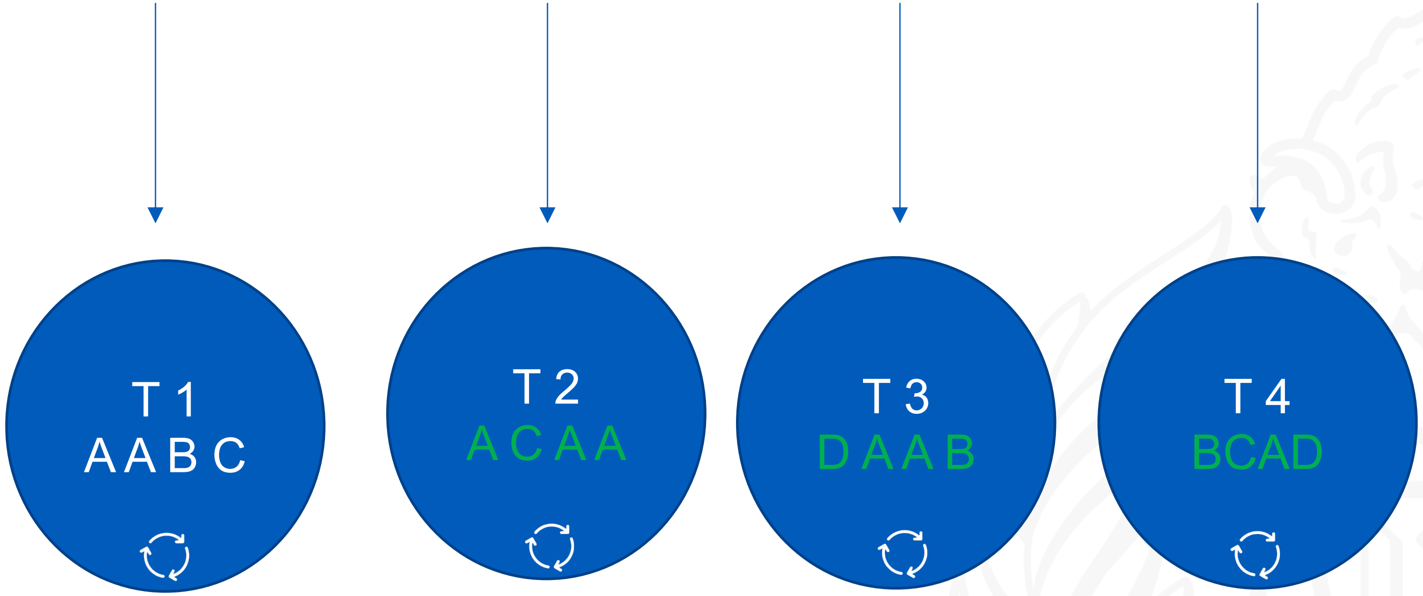
Pattern: D A A B

Pattern existence in two processors



Pattern: D A A B B C A D

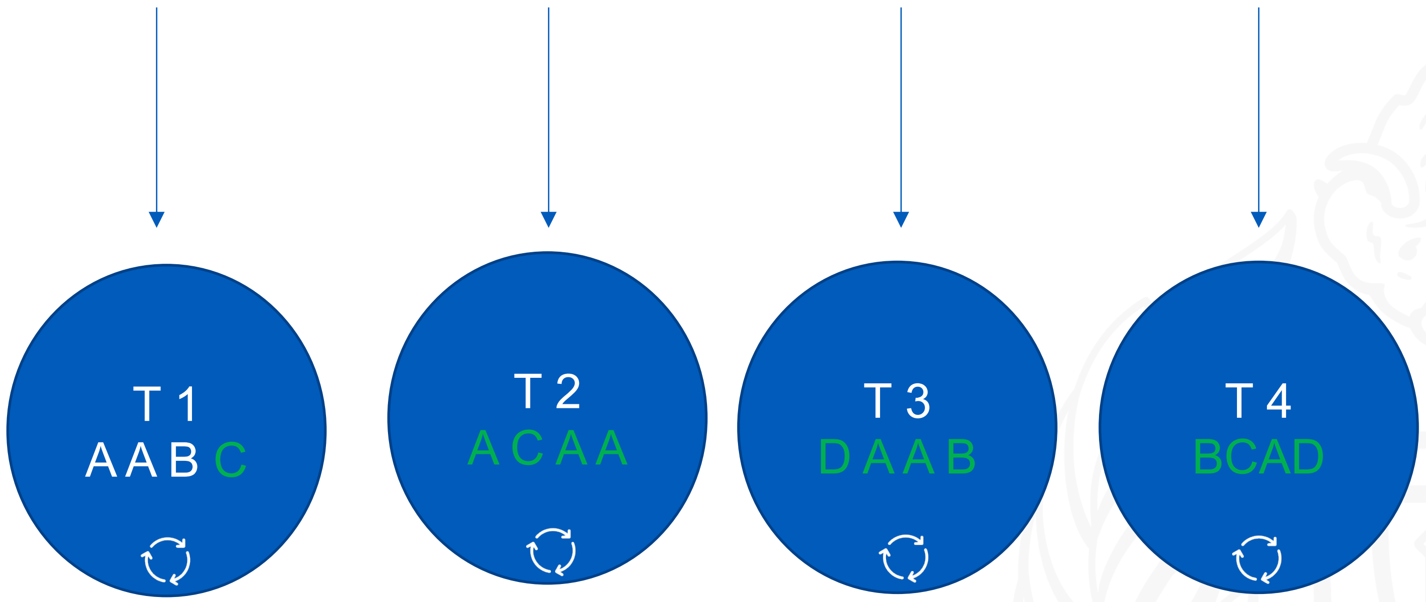
Pattern existence in three processors



Pattern: A C A A D A A B B C A D

Instance 1	0	4	0	0
Instance 2	0	4	8	0
Instance 3	0	4	8	12

Pattern existence in four processors

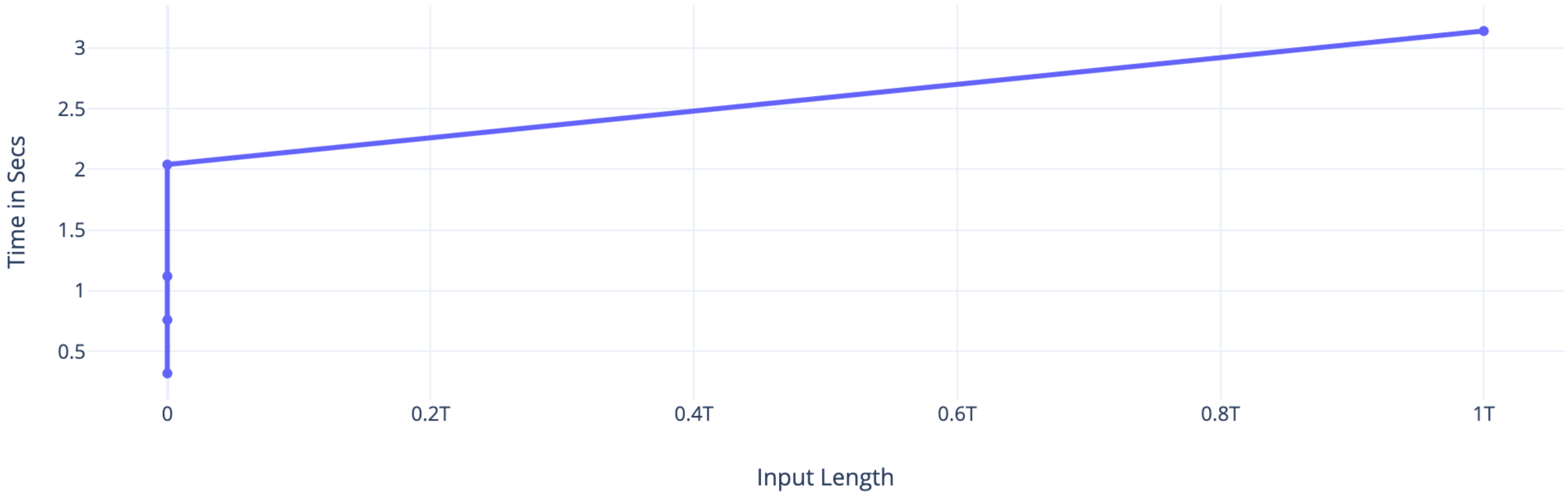


Pattern: C A C A A D A A B B C A D

Instance 1	1	0	0	0
Instance 2	1	5	0	0
Instance 3	1	5	9	0
Instance 4	1	5	9	13

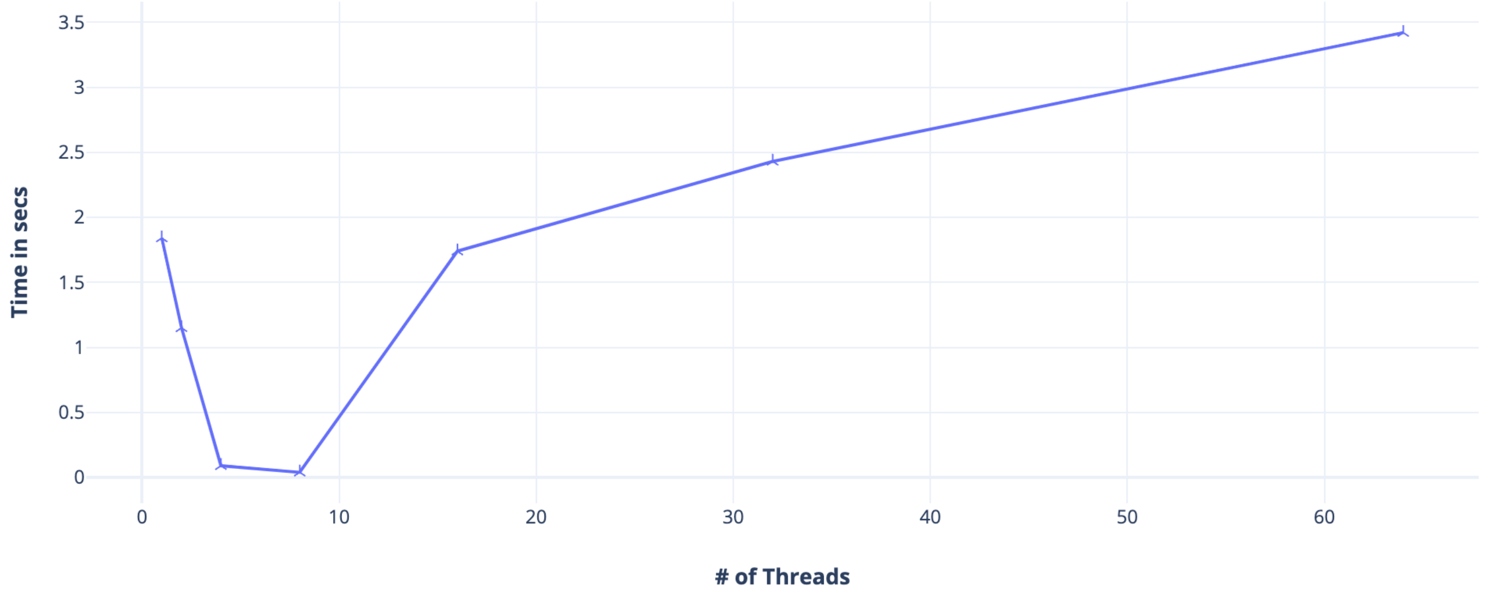
Input Size Vs Time In Secs for Sequential KMP

Input Len	Time in secs
1e3	0.32
1e4	0.76
1e5	1.12
1e6	2.04
1e12	3.14



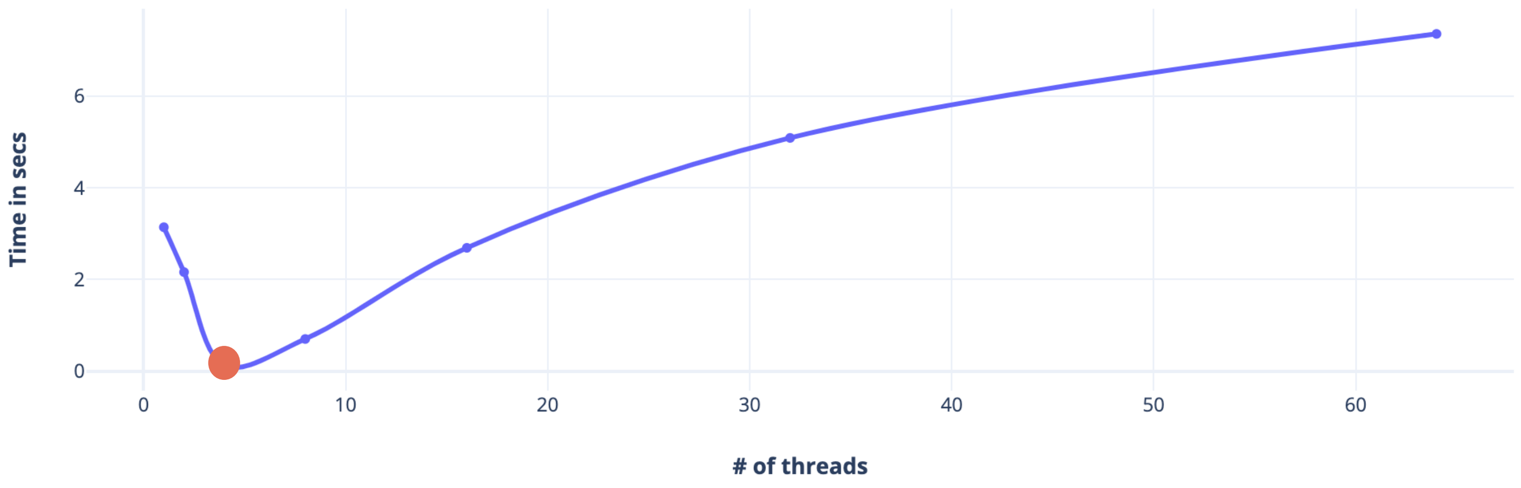
Threads Vs Run Time Text Size=1e6 & P=33 (Open MP)

Processors	Secs
1	1.84
2	1.15
4	0.09
8	0.04
16	1.74
32	2.43
64	3.42



Threads Vs Run Time Text Size=1e12 & P=33 (Open MP)

Processors	Secs
1	3.14
2	2.16
4	0.12
8	0.70
16	2.69
32	5.09
64	7.36



TCP/IP Vs IB|OPA Network Band Performance ?

It doesn't make any difference

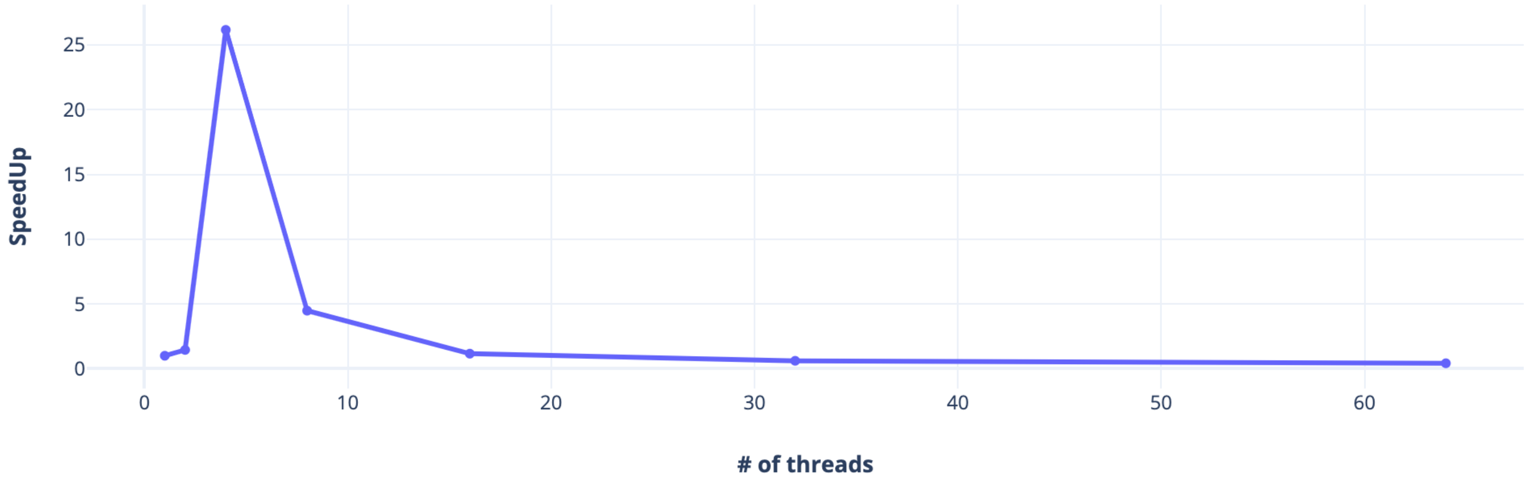
University at Buffalo The State University of New York

Speed Up Vs Processors

Input text size 1e12, T_{seq} = 3.14 secs

$$\text{SpeedUp} = \frac{T_{Seq}}{T_{Parallel}}$$

Threads	$T_{Parallel}$	Speed Up	Data Per Thread
1	3.14	1	1e12
2	2.16	1.45	500000000000
4	0.12	26.16	250000000000
8	0.70	4.48	125000000000
16	2.69	1.16	62500000000
32	5.09	0.61	31250000000
64	7.36	0.42	15625000000



Slurm Script

The slurm job script is designed to utilize the entire 1 node with 64 cores in ub-hpc cluster, where each thread would take one core to perform the computations.

$ kmp_openmp.slurm

kmp_openmp > $ kmp_openmp.slurm

```
#!/bin/bash

#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=64
#SBATCH --exclusive
#SBATCH --partition=general-compute
#SBATCH --qos=general-compute
#SBATCH --cluster=ub-hpc
#SBATCH --reservation=ubhpc-future

#SBATCH --time=00:10:00
#SBATCH --mail-type=END
#SBATCH --output=slurmOMP.out
#SBATCH --job-name=parallel-kmp

##export I_MPI_OFI_PROVIDER=sockets

module load intel


gcc -fopenmp -o parallel_kmp kmp.c -lm

## speedup runs
for nt in 1 2 4 8 16 32 64; do
export OMP_NUM_THREADS=$nt
./parallel_kmp
done
```

References

- https://ieeexplore.ieee.org/document/6618720
- https://cmps-people.ok.ubc.ca/ylucet/DS/KnuthMorrisPratt.html
- https://buffalo.app.box.com/s/nqj3neyt2w1dtb3gix6zxqx5gcc9x30n
- http://koreascience.or.kr/article/JAKO201814955686557.page
- https://ieeexplore.ieee.org/document/8599534

Thank You

Questions?