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)

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

Takes O(m) time to generate the LPS array
KMP Pattern Matching


Pattern: 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


Pattern: 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


Pattern: 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


Pattern: A A B A A A A B A A B 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 \( \text{len}(\text{string-1}) \) in the \( i \)th 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

Text: D A A B B C A D

Pattern: D A A B B C A D

LLPM Table

Pattern: D A A B B C A D

Pattern: D A A B B C A D

T 0
D A A B

T 1
B C A D
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 $\left(\frac{n}{k}\right)$ 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 $\left(\frac{n}{k}\right)$ 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 recomputes 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

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

0 1 2 3 4 5 6

Read cumulative LLP from its predecessor
Read non-cumulative LLP from its predecessor
Write cumulative LLP for its successor
Write non-cumulative LLP for its successor

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

<table>
<thead>
<tr>
<th>Instance</th>
<th>T1 (A ABC)</th>
<th>T2 (ACAA)</th>
<th>T3 (DAAB)</th>
<th>T4 (BCAD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance 1</td>
<td>0 4 0 0</td>
<td>0 4 0 0</td>
<td>0 4 0 0</td>
<td>0 4 0 0</td>
</tr>
<tr>
<td>Instance 2</td>
<td>0 4 8 0</td>
<td>0 4 8 0</td>
<td>0 4 8 0</td>
<td>0 4 8 0</td>
</tr>
<tr>
<td>Instance 3</td>
<td>0 4 8 12</td>
<td>0 4 8 12</td>
<td>0 4 8 12</td>
<td>0 4 8 12</td>
</tr>
</tbody>
</table>
### Pattern existence in four processors

<table>
<thead>
<tr>
<th>Instance</th>
<th>T1 (AABC)</th>
<th>T2 (ACAA)</th>
<th>T3 (DAAB)</th>
<th>T4 (BCAD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instance 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Instance 2</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Instance 3</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Instance 4</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>13</td>
</tr>
</tbody>
</table>

Pattern: C A C A A D A A B B C A D
Input Size Vs Time In Secs for Sequential KMP

<table>
<thead>
<tr>
<th>Input Len</th>
<th>Time in secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e3</td>
<td>0.32</td>
</tr>
<tr>
<td>1e4</td>
<td>0.76</td>
</tr>
<tr>
<td>1e5</td>
<td>1.12</td>
</tr>
<tr>
<td>1e6</td>
<td>2.04</td>
</tr>
<tr>
<td>1e12</td>
<td>3.14</td>
</tr>
</tbody>
</table>

![Graph showing input size vs time in seconds for sequential KMP]
Threads Vs Run Time Text Size=1e6 & P=33 (Open MP)

<table>
<thead>
<tr>
<th>Processors</th>
<th>Secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.84</td>
</tr>
<tr>
<td>2</td>
<td>1.15</td>
</tr>
<tr>
<td>4</td>
<td>0.09</td>
</tr>
<tr>
<td>8</td>
<td>0.04</td>
</tr>
<tr>
<td>16</td>
<td>1.74</td>
</tr>
<tr>
<td>32</td>
<td>2.43</td>
</tr>
<tr>
<td>64</td>
<td>3.42</td>
</tr>
</tbody>
</table>

![Graph showing the relationship between number of threads and run time.](image-url)
Threads Vs Run Time Text Size=1e12 & P=33 (Open MP)

<table>
<thead>
<tr>
<th>Processors</th>
<th>Secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14</td>
</tr>
<tr>
<td>2</td>
<td>2.16</td>
</tr>
<tr>
<td>4</td>
<td>0.12</td>
</tr>
<tr>
<td>8</td>
<td>0.70</td>
</tr>
<tr>
<td>16</td>
<td>2.69</td>
</tr>
<tr>
<td>32</td>
<td>5.09</td>
</tr>
<tr>
<td>64</td>
<td>7.36</td>
</tr>
</tbody>
</table>
TCP/IP Vs IB|OPA Network Band Performance?

It doesn’t make any difference
# Speed Up Vs Processors

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

<table>
<thead>
<tr>
<th>Threads</th>
<th>$T_{Parallel}$</th>
<th>Speed Up</th>
<th>Data Per Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14</td>
<td>1</td>
<td>$1e12$</td>
</tr>
<tr>
<td>2</td>
<td>2.16</td>
<td>1.45</td>
<td>500000000000</td>
</tr>
<tr>
<td>4</td>
<td>0.12</td>
<td>26.16</td>
<td>250000000000</td>
</tr>
<tr>
<td>8</td>
<td>0.70</td>
<td>4.48</td>
<td>125000000000</td>
</tr>
<tr>
<td>16</td>
<td>2.69</td>
<td>1.16</td>
<td>62500000000</td>
</tr>
<tr>
<td>32</td>
<td>5.09</td>
<td>0.61</td>
<td>31250000000</td>
</tr>
<tr>
<td>64</td>
<td>7.36</td>
<td>0.42</td>
<td>15625000000</td>
</tr>
</tbody>
</table>
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.
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?