Efficient Parallel Algorithms for String Comparison

Nikita Mishin, Daniil Berezun, Alexander Tiskin

JetBrains Research, Programming Languages and Tools Lab
Saint Petersburg State University, Russia

11.08.2021
Introduction::Longest Common Subsequence

LCS

- \( a = a_1a_2...a_m, \ b = b_1b_2...b_n \)
- \( LCS(a, b) = \) length of longest common subsequence
- \( a = CIPR, \ b = ICPP \rightarrow LCS(a, b) = LCS(CIPR, ICPP) = 2 \)
- \( a = BAABCBCA, \ b = BAABCABCABACA \rightarrow LCS(a, b) = LCS(BAABCBCA, BAABCBABCABACA) = 8 \)
- \( O(nm) \)
Informal definition:

- \( m + n \) monotone curves (called strands)
- Neighboring strands can form a crossing
- Neighboring strands can cross at most once

\[
\{0 \leftrightarrow 1, 1 \leftrightarrow 2, 2 \leftrightarrow 4, 3 \leftrightarrow 0, 4 \leftrightarrow 5, 5 \leftrightarrow 3\}
\]
Multiplication $O((m + n) \log(m + n))$ — place one braid under another and untangle strands
Introduction::semi-local LCS

\[
a = \text{BAABCBCA} \\
b = \text{BAABCABCABACA} \\
H[i, j] = \text{LCS}(a, b[i : j]) \\
H(4, 11) = 5
\]
Introduction::semi-local LCS

- Can be expressed via sticky braids objects of size $n + m$
  - Embeddings into LCS grid
    - Rotate braid by 45 degrees anti-clockwise
    - Symbol matches – barrier for strands to intersect
  - Two approach:
    - Divide-and-conquer: split into smaller braids; to concatenate apply sticky braid multiplication
    - DP: process cell-by-cell and cross strands if needed
  - $O(nm)$
Introduction::semi-local LCS

\[ m + n - 1 \]
Implementation::DP
if(a_symb = b_symb) || (h_strand > v_strand))

swap(h_strand, v_strand)
### Efficient string algorithms

#### Implementation::DP

![Diagram of string matching with DP approach]

<table>
<thead>
<tr>
<th>B</th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementation::DP

- ← and ↑ — cell dependency

- if(a_symb = b_symb) || (h_strand > v_strand))
  swap(h_strand, v_strand) — inside cell computation
Thread-level parallelization via antidiagonal pattern

SIMD parallelization via branch elimination:

\[
\begin{align*}
    h_{\text{strand}}' &= (h_{\text{strand}} \& (p - 1)) \mid ((-p) \& v_{\text{strand}}) \\
    v_{\text{strand}}' &= (v_{\text{strand}} \& (p - 1)) \mid ((-p) \& h_{\text{strand}})
\end{align*}
\]

Bonus №1: for \( m + n < 2^t \) t bits per strand sufficient

Bonus №2: possible load balancing through braid multiplication:
Implementation::Recursive
Implementation::Recursive
Implementation::Recursive
Implementation::Recursive

Implementation::Recursive

- core unit — steady ant algorithm:
  - Fast matrix multiplication $O(n \log n)$ (also divide-and-conquer)

- Deep recursion
Implementation::Recursive

- Processor-level parallelism

- Efficient memory management:
  - No malloc inside function
  - Reuse of space from outer levels

- Precalc product of permutations up to some $N$:
  - Small permutations fit to one machine word
  - $N! \times N!$ pairs for $N$
  - Lookup to map $\text{pair}(p, q)$
Eliminate outer recursion:

- Split into fixed-size subproblem: \( m_i + n_i < 2^{16} \)
- One thread per problem
- Then apply sticky braid multiplication in parallel fashion
Implementation::Bit-parallel prefix LCS

- Bit-parallel prefix LCS for binary strings:
  - Hyyrö, Crochemore et al.
  - Integer addition
  - Therefore, carry propagation
Implementation:: Bit-parallel prefix LCS for binary strings

Idea

- 1 for horizontal strands, 0 for vertical
- Most significant bit first for $a$ and horizontal strands
- Least significant bit first for $b$ and vertical strands
- Shifts for word alignment, Boolean operators for cell logic
- Process in antidiagonal tiles

$LCS(a, b) = |a| - \text{set bits in } h$
Example

- \( w = 4 \)

- \( a = 1000, \ b = 0100 \)

**Encoding:**
- \( a' = 1000_2, \ b' = 0010_2 \)
- \( h = 1111_2, \ v = 0000_2 \)
Implementation:: Bit-parallel prefix LCS for binary strings

\[ h = 1111 \]
\[ v = 0000 \]

\[ h = 0011 \]
\[ v = 1100 \]

\[ h = 0111 \]
\[ v = 1000 \]

\[ h = 1111 \]
\[ v = 1010 \]

\[ h = 0011 \]
\[ v = 1011 \]

\[ h = 0001 \]
\[ v = 1011 \]

\[ h = 0001 \]
\[ v = 1011 \]
Processing of second antidiagonal (18 op):

- Compare characters: \( s = \neg((a' \gg 2) \oplus b) \)
- Active bits: \( mask = 0011_2 \) (compile time)
- Combing condition: \( c = mask \& (s \mid (\neg(h \gg 2) \& v)) \)
- save \( v' = v \)
- update \( v = (\neg c \& v) \mid (c \& (h \gg 2)) \)
- update \( c = c \ll 2 \)
- update \( h = (\neg c \& h) \mid (c \& (v' \ll 2)) \)
Processing of second antidiagonal (18 op):

- Compare characters: $s = !((1000_2 \gg 2) \oplus 0010_2) = 0011_2$

- Active bits: $mask = 0011_2$ (compile time)

- Condition: $c = 0011_2 \land (0011_2 \lor (!((1111_2 \gg 2) \& 0000_2)) = 0011_2$

- $v' = 0000_2$

- $v = (!0011_2 \& 0000_2) \lor (0011_2 \& (1111_2 \gg 2)) = 0011_2$ (1100)

- $c = 0011_2 \ll 2 = 1100_2$

- $h = (!1100_2 \& 1111_2) \lor (1100_2 \& (0000_2 \ll 2)) = 0011_2$ (0011)
Optimizations

- Register usage

- Update Rule optimization:
  - !a
Processing of second antidiagonal (11 op):

- Compare string characters: \( s = ((a'' \gg 2) \oplus b) \)

- \( v' = v \)

- \( v = ((h \gg 2) | \neg \text{mask}) \& (v \mid (s \& \text{mask})) \)

- \( h = h \oplus (v \ll 2) \oplus (v' \ll 2) \).
Evaluation::Main results

- AMD Ryzen-7-3800X, 8 cores and 16 threads, C++, G++ 10.2.0
- Synthetic dataset for different matching frequency:
  - $\sigma = 1$ — High
  - $\sigma = 5$ — Medium
  - $\sigma = 26$ — Low
- Real-data: Genome of viruses from NCBI
Evaluation::Main results

- **Nikita Mishin et.al. (SPBU)**
- **Efficient string algorithms**
Evaluation::Main results

strings of equal lengths $\sigma = 1$

strings of equal lengths $\sigma = 26$

Viruses of appx. equal lengths
Evaluation::Main results

\[ \sigma = 1, \ m = n = 100000 \]

\[ \sigma = 26, \ m = n = 100000 \]

Viruses, \( m = 124884, \ n = 134226 \)
Evaluation::Main results

Memory access optimization

Boolean formula optimization

Nikita Mishin et.al. (SPBU)
Relative performance of bit-parallel algorithm against semi-local LCS
Semi-local LCS (theory works in practice!)

Hybrid approach for semi-local LCS

Bit-parallel prefix LCS without adders based on sticky braid
Semi-local LCS is cool
Let’s study it!