Parallelizing Crochemore’s Repetitions Algorithm to Compute Runs in Strings

Mei Jiang

Advanced Optimization Laboratory
Department of Computing and Software
McMaster University

April 27, 2010
1 Outline

2 Introduction

3 Parallelization of FSX03

4 Parallelization of C2-K

5 Summary & Future Work
A run, a maximal fractional repetition in a string was conceptually introduced by Main in 1989[4].


A typical linear time algorithm for computing runs: suffix array $\Rightarrow$ L-Z factorization $\Rightarrow$ leftmost runs $\Rightarrow$ all runs

The linear-time algorithms for computing runs are not very conducive to parallelization mainly because the suffix tree or suffix array rely on recursion.

Though Crochemore’s repetitions algorithm has complexity of $O(n \log n)$, its strategy of repeated refinements of classes of equivalence, a process can be naturally parallelized.
Repetition

**Definition**

\((s, p, e)\) is a repetition in \(x\) iff
\[
x[s+i] = x[s+p+i] = \cdots = x[s+(e-1)p+i] \text{ for } 0 \leq i < p \text{ and } e \geq 2.
\]
s is the starting position, \(p\) is the period, \(e\) is the exponent (or power), and \(x[s..s+p-1]\) the generator of the repetition.
The generator must be irreducible (not a repetition).

The repetition can be encoded as \((s, p, d)\), where \(d\) is the ending position of the repetition, with \(d = s + ep - 1\).
Definition

\((s, p, e, t)\) is a run in \(x\), if

1. for every \(0 \leq i \leq t\), \((s + i, p, e)\) is a maximal repetition, and
2. either \(s = 0\) or \(x[s - 1] \neq x[s + p - 1]\) (the run cannot be extended to the left), and
3. either \(s + ep + t > n\) or \(x[s + (e - 1)p + t] \neq x[s + ep + t]\) (the run cannot be extended to the right).

\[
x = b\ a\ b\ a\ a\ a\ b\ a\ a\ a\ b\ b
\]

A run \((s, p, e, t)\) can be encoded as \((s, p, d)\) where \(d\) is the end position of the run, with \(e = (d - s + 1)/p\) and \(t = (d - s + 1)\%p\).
Crochemore’s Repetitions Algorithm

1981 Crochemore designed the first $O(n \log n)$ algorithm to compute all the repetitions in a string[3]. The main ideas of his approach is to refine the indices of the string into several equivalent classes at each level.

We say two indices at level $l$ are equivalent if two identical substring of length $l$ start there. i.e. $f = abcab \{1, 4\}_{ab}$ at level 2

After initial refinement, the original input string need not be accessed anymore, the rest refinements use other classes from previous level and only those so-called small classes, which brings the worst-case complexity to $O(n \log n)$.
Parallelizing Crochemore’s Alg. to Compute Runs in Strings

Crochemore’s Repetitions Algorithm - Example

f = a b a a b a a b a b a $

$l=1$

{1, 3, 4, 6, 7, 9, 11}$_a$

{2, 5, 8, 10}$_b$

{l=2}

{1, 4, 7, 9}$_{ab}$

{3, 6}$_{aa}$

{11}$_{a}$

{2, 5, 8, 10}$_{ba}$

{l=3}

{1, 4, 7, 9}$_{aba}$

{3, 6}$_{aab}$

{2, 5}$_{baa}$

{8}$_{bab}$

{10}$_{baa}$

{l=4}

{1, 4}$_{abaa}$

{7}$_{ab}$

{9}$_{aba}$

{3, 6}$_{aaba}$

{2, 5}$_{baab}$

{l=5}

{1, 4}$_{abaab}$

{3}$_{aabaa}$

{6}$_{aab}$

{2, 5}$_{baaba}$

{l=6}

{1, 4}$_{abaaba}$

{2}$_{baabaa}$

{5}$_{baabab}$

{l=7}

{1}$_{abaabaa}$

{4}$_{abaabab}$
Franek et. al. gave a most memory efficient implementation of Crochemore’s algorithm referred to as **FSX03**[2].

In [1], algorithm C was the best extension algorithm to compute in terms of performance though it requires an extra $O(n \log n)$ memory space. Its variant **C2-K** was introduced to reduce the memory requirement.

Parallelize FSX03 & C2-K to compute runs within **shared memory model**.
FSX03 Overview

- FSX03 implements the refinement step by traversing and processing the all the elements in the small classes. For each element $e$, $e - 1$ gets refined.

- Refine current level of classes from previous level of classes. However it’s too expensive to keep two levels, a notion of “snapshot” is used to keep the small classes from previous level.

- When refine a class it involves moving the element from its original class to a new class or leaving it in place. FSX03 uses Refine[] and RefStack[] to keep track of these classes. They are cleared after processing each small class.
**FSX03 Overview**

Alternative 1

Alternative 2

**Remark**

---

**Small Classes:** \{e_{11}, e_{12}, \ldots\} \{e_{21}, e_{22}, \ldots\} \ldots

**Classes:** C_i = \{e_{11} -1, \ldots\} C_j = \{e_{12} -1, \ldots\} \ldots

C_{k1} = \{e_{11} -1\} C_{k2} = \{e_{12} -1\}

---

**SCQueue[]**

| e_{11} | e_{21} |

**SElQueue[]**

| e_{11} | e_{12} | \ldots | e_{21} | \ldots |

**RefStack[]**

| i | j |

**Refine[]**

| k1 | k2 |

---

**Mei Jiang**

Parallelizing Crochemore’s Alg. to Compute Runs in Strings
Alternative 1

Assign each small class to a processor to process refinements simultaneously.

- Extra memory for `Refine[]` and `RefStack[]` required for each processor.
- Less processors required.
Allocate memory for \texttt{Refine[]} and \texttt{RefStack[]}:

- **Static**: size of $n$ for both
- **Dynamic**: size of the assigned small class to \texttt{RefStack[]} and the largest class number to \texttt{Refine[]}

Processor 1
\[ S_1 = \{e_{11}, e_{12}, \ldots, e_{1k}\} \]

Processor 2
\[ S_2 = \{e_{21}, e_{22}, \ldots, e_{2i}\} \]

Processor t
\[ S_t = \{e_{t1}, e_{t2}, \ldots, e_{tj}\} \]

Master
Processor
Each class is refined by all small classes, assign the refinement of each class to a processor.

- No extra memory required.
- More processors required.
Refine every class by all the small class: $S_1$, $S_2$, ...

Diagram:

- Master Processor
- Processor 1: $C_1 \times S_1$
- Processor i: $C_1 \times S_2$
- Processor j: $C_1 \times S_3$
- Processor t: $C_t \times S_1$
- Processor i': $C_t \times S_2$
- Processor j': $C_t \times S_3$

Mei Jiang
Parallelizing Crochemore’s Alg. to Compute Runs in Strings
Remark

- Mutual exclusion locking for both read and write required for critical routines i.e. AddToClass or RemoveFromClass.
- Other steps in FSX03 could potentially be parallelized. i.e. computation of the level 1 can be done by partitioning the string and processed by multiple processors.
C Overview

C is an extension algorithm to compute runs:

1. **Collect** all the repetitions into an array of linked lists based on their starting positions.
2. **Traverse** all the repetitions and consolidates the “nearby” repetitions with the same period into runs.
C2-K Overview

C2-K is a variant of C, and it’s designed for bringing down the memory requirement of C.

1. Partially consolidates repetitions into runs when putting them into the buckets. For a repetition with period $p \leq K$ and start $s$, we check $p$ buckets to the left and to the right of $s$; for $p > K$, we check $K$ buckets.

2. Traverses and consolidates the repetitions with periods $p > K$ as C2-K guarantees that all repetitions up to period $K$ have been consolidated into runs before the final sweep.
Description

- Break down consolidation work in terms of periods and each processor is assigned with a range of periods. Every processor traverse the buckets and consolidate only the repetitions with assigned periods.
- The range of the periods of string for C2-K is \((K + 1, \lceil n/2 \rceil)\).
  - Equally distributes over \(P\) processors. \(\lceil (n/2 - (K + 1) + 1)/P \rceil\) number of periods are assigned to each processor.
  - Or assign a fixed number periods \(t\) to each processors until all the periods have been done.
There is **NO** extra data structure required for the parallelizing C2-K.

There is **NO** extra data structure required for the parallelizing C2-K.
Remark

- No additional space is required.
- No extra actions such as locking are needed.
- Might increase the overall complexity, however, overall execution time should not be affected.
We have investigated parallelization of Crochemore’s repetitions algorithm to compute runs within **shared memory model**.

We are currently working on **implementation** for a multi-core machine platform and extensive **testing** against various types of strings.

We intend to investigate all aspects of parallelization of the extended Crochemore’s algorithm within **distributed memory model**.

We plan on using **SHARCNET** as the hardware platform for the implementation of the distributed memory parallel version of the algorithm.
References


THANK YOU!