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

Background

- A run, a maximal fractional repetition in a string was conceptually introduced by Main in 1989[4].
- R. Kolpakov and G. Kucherov[5] showed how to compute all the runs from leftmost runs in 2000.
- A typical linear time algorithm for computing runs:
  suffix array $\Rrightarrow$ L-Z factorization $\Rrightarrow$ leftmost runs $\Rrightarrow$ 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* $\mathbf{x}[s+i] = \mathbf{x}[s+p+i] = \cdots = \mathbf{x}[s+(e-1)p+i]$ for $0 \leq i < p$ 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).

		1	2	3	4	5	6	7	8	9	10
x	$=$	b	**a**	**b**	**a**	**a**	**b**	**a**	a	b	b

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

Run

Definition

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

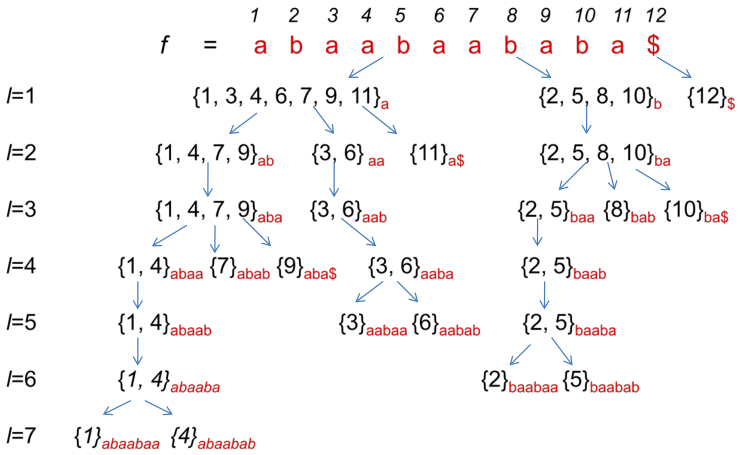
1. for every $0 \le i \le t$, $(s+i, p, e)$ is a maximal repetition, and
2. either $s = 0$ or $\mathbf{x}[s-1] \neq \mathbf{x}[s+p-1]$ (the run cannot be extended to the left), and
3. either $s + ep + t > n$ or $\mathbf{x}[s+(e-1)p+t] \neq \mathbf{x}[s+ep+t]$ (the run cannot be extended to the right).

		1	2	3	4	5	6	7	8	9	10
x	$=$	b	**a**	**b**	**a**	**a**	**b**	**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 $O(n \log n)$.

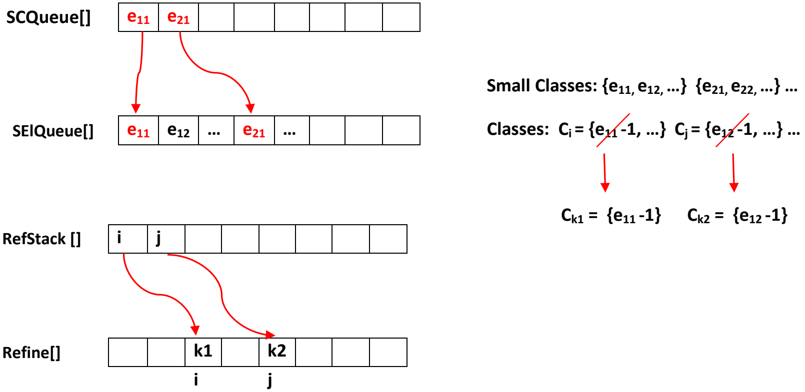

1 2 3 4 5 6 7 8 9 10 11 12
f = a b a a b a a b a b a $
l=1
{1, 3, 4, 6, 7, 9, 11}a
{2, 5, 8, 10}b
{12}$
l=2
{1, 4, 7, 9}ab
{3, 6} aa
{11}a$
{2, 5, 8, 10}ba
l=3
{1, 4, 7, 9}aba
{3, 6}aab
{2, 5}baa
{8}bab
{10}ba$
l=4
{1, 4}abaa
{7}abab
{9}aba$
{3, 6}aaba
{2, 5}baab
l=5
{1, 4}abaab
{3}aabaa
{6}aabab
{2, 5}baaba
l=6
{1, 4}abaaba
{2}baabaa
{5}baabab
l=7
{1}abaabaa
{4}abaabab

FSX03 & C2-K

- 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

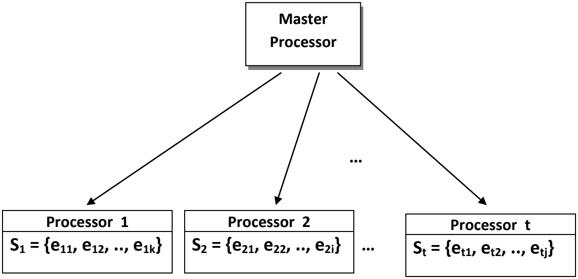
- FSX03 implements the refinement step by traversing and processing the all the elements in the small classes. For each element e, $e-1$ gets refined.
- Refine current level of classes from previous level of classes. However it's too expensive to keep two levels, a notion of "snapshot" is used to keep 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.


SCQueue[]
e11
e21
SElQueue[]
e11
e12
...
e21
...
RefStack []
i
j
Refine[]
k1
k2
i
j
Small Classes: {e11, e12, ...} {e21, e22, ...} ...
Classes: Ci = {e11 -1, ...} Cj = {e12 -1, ...} ...
Ck1 = {e11 -1}
Ck2 = {e12 -1}

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 `Refine[]` and `RefStack[]`:

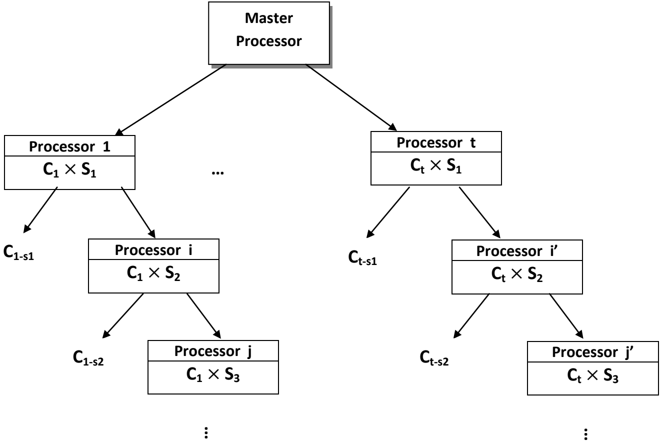
- **Static**: size of n for both
- **Dynamic**: size of the assigned small class to `RefStack[]` and the largest class number to `Refine[]`

Alternative 2

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, ...



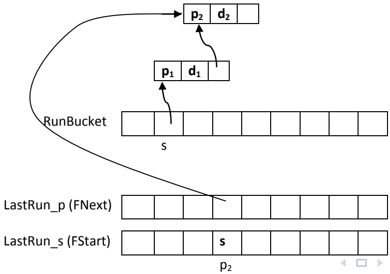
Remark

- 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 a 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 bring down the memory requirement of C.

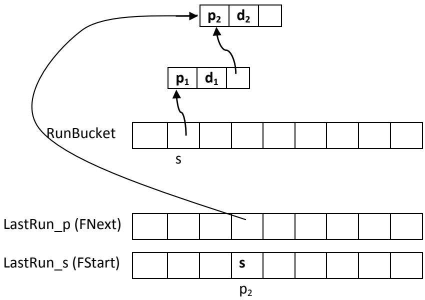
1. Partially consolidates repetitions into runs when putting them into the buckets. For a repetition with period $p \leq K$ and start s, we check p buckets to the left and to the right of s; for $p > K$, we check K buckets.
2. Traverses and consolidates the repetitions with periods $p > K$ as C2-K guarantees 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 ranges 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 $(K+1, \lfloor n/2 \rfloor)$.
  - 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.

Data Structure

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.

Summary & Future Work

- We have investigated parallelization of Crochemore's repetitions algorithm to compute runs within **shared memory model**.
- We are currently working on **implementation** for a multiple-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

- F. Franek and M. Jiang, *Crochemore's repetitions algorithm revisited - computing runs*, Proceedings of Prague Stringology Conference, (2009), pp. 123-128.
- F. Franek, W. F. Smyth and X. Xiao, *A note on Crochemore'srepetitions algorithm a fast space-efficient approach*, Nordic Journal of Computing, 10(2003), pp. 21-28.
- M. Crochemore, *An optimal algorithm for computing the repetitions in a word*, Inform. Process. Lett., 125(1981), pp. 244-250.
- M. G. Main, *Detecting leftmost maximal periodicities*, Discrete Applied Maths., 25(1989), pp. 145-153.
- R. Kolpakov and G. Kucherov, *On maximal repetitions in words*, J. Discrete Algs., 1(2000), pp. 159-186.

THANK YOU!