McMaster University

Advanced Optimization Laboratory

Title:

A Parallel Approach to Computing Runs in a String

Authors:

Frantisek Franek, and Mei Jiang

AdvOL-Report No. 2010/5

June 2010, Hamilton, Ontario, Canada

A Parallel Approach to Computing Runs in a String

Frantisek Franek, and Mei Jiang *

Advanced Optimization Laboratory, Dept. of Computing and Software
McMaster University, Hamilton, Ontario, L8S 4K1 Canada
email: {franek,jiangm5}@mcmaster.ca

Abstract. The total number of runs in a string can be computed using the Lempel-Ziv factorization obtained from the suffix tree or array of the string. A number of linear-time algorithms have been developed based on this strategy over the last few years. These algorithms are not very conducive to parallelization, in particular the recursion-based linear-time computation of the suffix tree or array. Recently, we introduced several extensions of Crochemore's repetitions algorithm for computing runs. Among these various extensions, we identified the one with the best performance and relatively good space efficiency. Though of complexity $O(n \log n)$, where n is the length of the input string, the nature of Crochemore's repetitions algorithm lends itself naturally to parallelization. In this paper, we propose a parallel approach to computing runs based on a parallelization of the extended Crochemore's algorithm under the shared memory model.

1 Introduction

The periodicity (repetitions of substrings) is the most studied and important topic in *combinatorics on words* and *algorithms on strings*, going back to Thue [11]. It has been applied in many different fields such as data mining, pattern-matching, data compression, and computational biology.

In [3], Crochemore introduced the first $O(n \log n)$ algorithm to compute all the repetitions in a string. The notion of runs that captures maximal fractional repetitions was introduced by Main in [10], where it was shown how to compute all leftmost runs from Lempel-Ziv factorization of the input string. [9] introduced an algorithm to compute all runs from the leftmost ones. Consequently, a number of linear-time algorithms have been developed for computing runs; all these algorithms are derived from Main's original idea and they are rather involved and complex. For their linearity, they rely on linear-time computation of the suffix tree or array. In comparison, Crochemore's algorithm is simpler and mathematically more elegant. The strategy of Crochemore's algorithm relies on repeated refinements of equivalence classes, a process that can be naturally parallelized, as the refinement of each class is independent from other classes' refinements and

* corresponding author

can be performed simultaneously by different processors in parallel. The linear-time algorithms for computing runs on the other hand are not very conducive to parallelization mainly because the linear-time computation of the suffix tree or array relies on recursion. To date, there have not been many attempts to parallelize computations of repetitions or runs, [1, 2, 4, 7, 8].

In [5], we introduced a number of extensions of Crochemore's repetitions algorithm for computing runs. One of them, C2-K was identified as the best in terms of performance and memory usage. C2-K was developed and tested based on the most space-efficient implementation of Crochemore's algorithm FSX03 [6]. In this paper, we discuss two approaches to parallelization of FSX03 and C2-K in a shared memory multiprocessor model.

2 Basic Notation

A ***repetition*** (i.e. a *tandem repeat*) in a string x, denoted as a triple (s, p, e), where $s \geq 1$ is the *starting* position, $p \geq 1$ the *period*, and $e \geq 2$ the *exponent* (or *power*), describes the fact that $x[s{+}i] = x[s{+}p{+}i] = \cdots = x[s{+}(e{-}1)p + i]$ for every $0 \leq i < p$. A maximal non-extendible repetition is such that the *generator* $x[s..s{+}p{-}1]$ itself is not a repetition, and the repetition cannot be extended to the left or to the right. For instance, $x = ababa$ has two maximal non-extendible repetitions: $(1, 2, 2)$ representing $(ab)^2$, and $(2, 2, 2)$ representing $(ba)^2$.

The concept of a ***run*** was first introduced by Main in [10], where it was called ***maximal periodicity***. The term ***maximal repetition*** is used in [9]. A run is formed by a maximal non-extendible repetition followed by a proper prefix (possibly empty) of the generator of the repetition, the so-called ***tail*** of the run. For example, $x = abaabaabaab = (aba)^3ab$ (formed by repetition $(aba)^3$ followed by a prefix of the generator ab) is a run of period $p = |aba| = 3$. Moreover, it is required that the run is maximal in that it is not extensible to the left or to the right. Runs can be seen as compressed forms of repetitions. One run thus encodes $t{+}1$ maximal non-extendible repetitions. Obviously, by listing all the runs, we also list all the repetitions of the string (of course implicitly), but with much less space.

A run can be encoded as (s, p, e, t); where s, p, e have the same meaning as for repetitions, and $0 \leq t < p$ is the tail. Alternatively, to be more space efficient, we could encode a run as a triple (s, p, d); where $d = s{+}(p \times e){+}t{-}1$ specifies the ending position of the run, and the exponent and the tail size of the run can be easily computed through equations $e = (d{-}s{+}1)/p$ (integer division) and $t = (d{-}s{+}1)\%p$ (modulus operation), respectively. Consider the following example:

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

Substring $x[2..9]$ of x specifies a run $(2, 3, 9)$, and it includes three repetitions $(aba)^2$, $(baa)^2$, and $(aab)^2$.

3 Crochemore's Repetitions Algorithm

In 1981, Crochemore designed the first $O(n \log n)$ algorithm to compute all maximal non-extendible repetitions in a string [3]. The main idea of this approach is to successively group the indices of the string into equivalence classes and refine them to smaller ones. Every equivalence class represents all repeats of a particular substring; in Figure 1, these substrings are indicated by the grey subscript of each class.

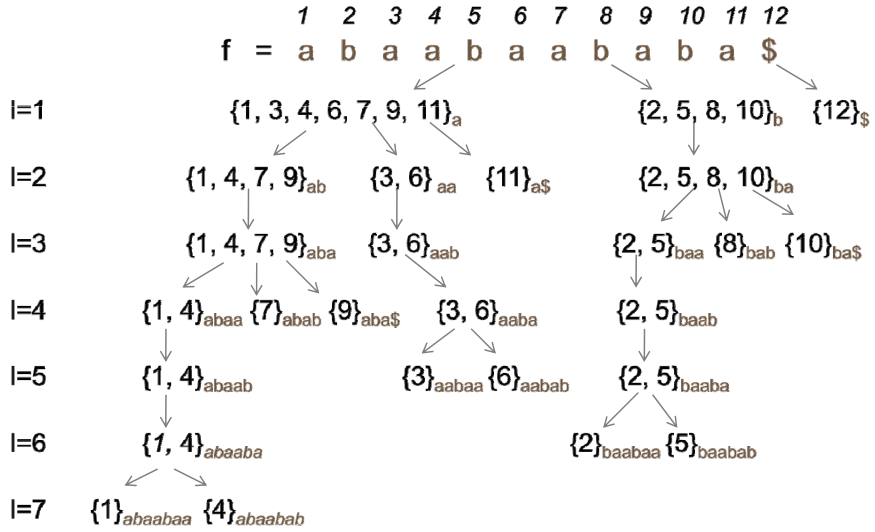

Fig. 1. Crochemore's repetitions algorithm - refinements of equivalence classes

For an input string $x[1..n]$, an equivalence $\approx_p$ on positions $\{1, 2, ..., n\}$ is defined by $i \approx_p j$ if $x[i..i+p-1] = x[j..j+p-1]$. For technical reasons, a sentinel symbol \$ is used to denote the end of the input string; it is considered the lexicographically smallest character. As shown in Figure 1, the indices of the input string have been grouped into classes of $\approx_p$, $p = 1..7$. Each class that has two or more elements represents a repeat of period p, where p equals the current level of refinement l.

Starting from level 1, Crochemore's algorithm continuously refines the equivalence classes until all classes have been reduced to a singleton (i.e. classes containing only a single element). An important aspect of Crochemore's algorithm

is the fact that the refinement of each equivalence class at each level is not carried out by investigating the original input string, but rather, it is refined using all the classes of the previous level. Note that the individual levels do not to be saved, all we need is the previous level to compute the new one. Consider a class $\mathcal{C}$ and a class $\mathcal{D}$. In order to refine class $\mathcal{C}$ by $\mathcal{D}$, for every $i, j \in \mathcal{C}$, if $i{+}1, j{+}1 \in \mathcal{D}$, then i, j are placed into the same class, otherwise they must be placed into different classes. To illustrate, let us refine a class $\mathcal{C} = \{1, 3, 4, 6, 7, 9\ 11\}$ by a class $\mathcal{D} = \{2, 5, 8, 10\}$ on level 1 (see Figure 1). The positions 1 and 3 must be placed in different classes as 2 and 4 are not both in $\mathcal{D}$, while 1 and 4 will be in the same class, since 2 and 5 are both in $\mathcal{D}$. In fact, using all classes for refinement, $\mathcal{C}$ will be refined into three classes, $\{1, 4, 7, 9\}$, $\{3, 6\}$,and $\{11\}$. When all classes are refined using all the classes from the previous level, the current level is computed.

Note that this process of refinement if carried out as described above would lead to an $O(n^2)$ algorithm. The most important aspect of Crochemore's algorithm is not to use all the classes for refinements, but only those so-called small classes. We call the classes that result from the refinement of a class, a *family*. For instance, in Figure 1, classes $\{1, 4, 7, 9\}$, $\{3, 6\}$ and $\{11\}$ on level 2 form a family as they are the classes obtained by the refinement of class $\{1, 3, 4, 6, 7, 9\ 11\}$ on level 1. For each family, we identify the largest class and call all the other classes in the family small. On level 1, all classes are designated as small. By using only small classes for refinement, $O(n \log n)$ complexity is achieved as each element belongs to at most $O(\log n)$ small classes.

As mentioned previously, every equivalence class represents all repeats of a particular substring. In order to report repetitions, each class needs to be examined for the "gaps" between the indices. If the gap is bigger than p, it indicates that the repeats are not tandem; if the gap is smaller than p, it indicates that the repeats are overlapped. Consider Figure 1, at level 3 in the class $\{1, 4, 7, 9\}_{aba}$ that represents all the repeats of *aba*, the gap between index 1 and 4 is 3 which equals the period (level) 3, and the gap between 4 and 7 is also 3; but the gap between 7 and 9 is 2 which is smaller than 3, indicating overlapping repeats. Thus, when considering repetitions, we should consider indices 1, 4, and 7, but not 9, which gives us the final output (1, 3, 3) representing the repetition $(aba)^3$ in the string.

4 Parallelization in the Shared Memory Model

4.1 Overview

With the shared memory parallel programming model, large tasks are decomposed into smaller subtasks, and the subtasks are assigned to multiple processors to compute simultaneously, while these processors share the same primary memory. Many contemporary machines with multi-core processors are the prime examples of a hardware realization of this model.

When designing an algorithm under this model, extra care must be taken to avoid corruption of the shared data and/or race conditions. In the following

sections, we discuss the ways to parallelize both, FSX03 [6] and its extension C2-K [5].

4.2 Parallelization of FSX03

FSX03 Overview Any implementation of Crochemore's repetitions algorithm requires data structures to track classes, small classes, families, and gaps. Any straightforward implementation of the refinement process (without tracking the gaps) requires around $20n$ integers, where n is the length of the input string. Using multiplexing (memory sharing) and some other tricks, FSX03 [6] managed to implement it using only $10n$ integers. Care must be taken to avoid traversing any of these structures in order to preserve the $O(n \log n)$ complexity.

Description of the First Approach The most straightforward approach to parallelizing FSX03 under the shared memory model, is to parallelize its core step, that is, the refinement of each level. Due to the nature of Crochemore's algorithm, each class at level l is refined by all small classes at the same level into new classes of level $l+1$. FSX03 implements it by traversing all small classes and using their elements for the refinements. Each of these refinements by a small class can be carried out independently; intuitively, the refinement by each small class can be assigned to a different processor (see Figure 2).

FSX03 does not keep the previous level while computing the current level, as this would require too much memory (the classes are stored in multiple doubly-linked lists). Instead a much less memory demanding "snapshot" of the previous level is created; the previous level becomes the (yet unfinished) current level and is modified through refinements until it becomes a finished current level.

At a level l, all the small classes are stored as a single "snapshot" in two queues `SElQueue` and `SCQueue`. The queue `SElQueue` stores the elements of the small classes in their natural order producing a list of all elements of all small classes. `SCQueue` stores the first element of each class with the same order with respect to `SElQueue`, thus providing the means to determine the beginning and end of each small class in `SElQueue`.

In the shared memory model, the master processor assigns each small class to a slave processor to process the elements in that class independently. This is possible because of the following two observations:

1. The purpose of the "snapshot" queues is to preserve the elements of the small classes from the previous level, so when a class c changes (has an element removed or added), this does not affect possible future refinements by c. Thus, when multiple processors process the various small classes in parallel, they will not interfere with each other.
2. When an element e from a small class is processed, it involves moving the element $e-1$ from one class to another or leaving it in its original class. When multiple processors process multiple elements at the same time, it is not hard to observe that no two processors are trying to move the same

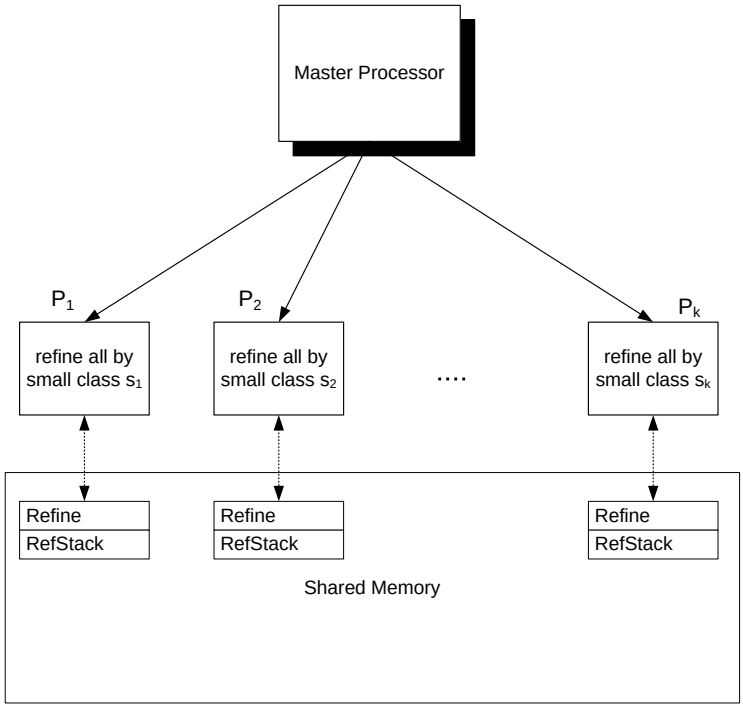

Fig. 2. Parallelization, first approach

element at the same time as the classes are always disjoint (each level is a partitioning of all indices, level l+1 is a refinement of level l). For this reason, the equivalence classes at level l can be refined by multiple processors into level l+1 in parallel without worrying about possible race conditions.

Data Structure As mentioned previously, when an element e from a small class is processed, it involves moving the element $e-1$ from its original class to a new class or leaving it in place. FSX03 uses an array `Refine` and a stack `RefStack` to keep track of these assignments. After it has finished processing all the elements in one small class, the contents of `Refine` and `RefStack` are purged by setting everything to NULL (note that this cannot be done by traversing the arrays as this would destroy the $O(n \log n)$ complexity), so when the program moves to the next small class, `Refine` and `RefStack` can be safely reused. This reuse implementation works rather well for the serial program as every element is processed sequentially and `Refine` and `RefStack` can be cleared out after each small class has been processed.

However, for this parallel implementation, every processor needs its own version of `Refine` and `RefStack` to manage this process to avoid conflicts with the other processors. To see this, let us discuss the situation when two processors are trying to modify the same class: consider two small classes s_i and s_j that are assigned and processed by two processors P_i and P_j, $e_i \in s_i$, $e_j \in s_j$ and $e_i{-}1, e_j{-}1 \in c_k$. When P_i processes the element e_i and $e_i{-}1$ is moved from the class c_k to a class c_h, this fact is recorded as `Refine[`k`]` $= h$. When P_j processes the element e_j in s_j and $e_j{-}1$ is moved from the class c_k to a class $c_{h'}$, this requires `Refine[`k`]`$= h'$. If the two processors shared the same `Refine` and `RefStack`, P_j would end up getting the class c_h as the recipient of $e_j{-}1$ to (as `Refine[`k`]`$= h$), which would be incorrect.

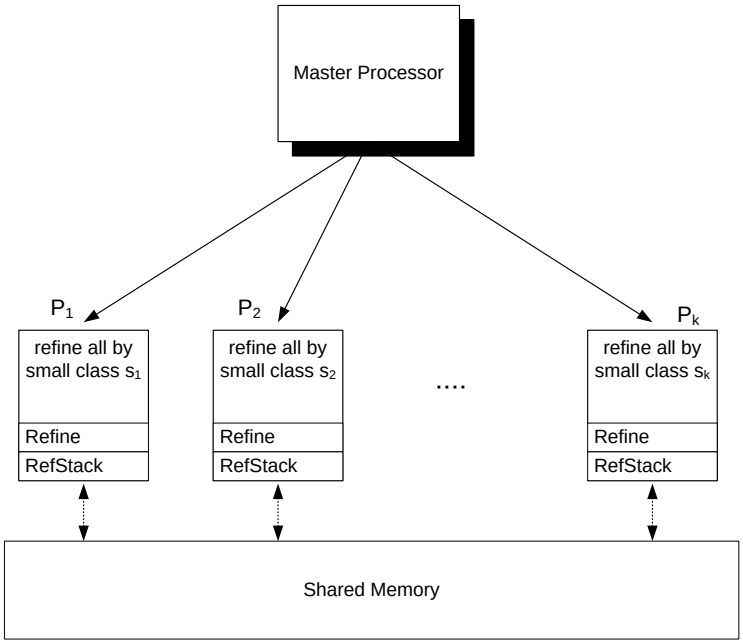

Fig. 3. Modified first approach

Therefore, an independent set of `Refine` and `RefStack` arrays must be allocated to each processor. This preserves the fast processing of FSX03, but requires a significant increase of memory usage as there would have to be P distinct versions of `Refine` and `RefStack`, where P is the number of processors available.

Nevertheless, there is a significant speed-up in processing, as in the serial version the program has to go through the whole "snapshot", while in the parallel version each processor just goes through the "snapshot" of a single small class.

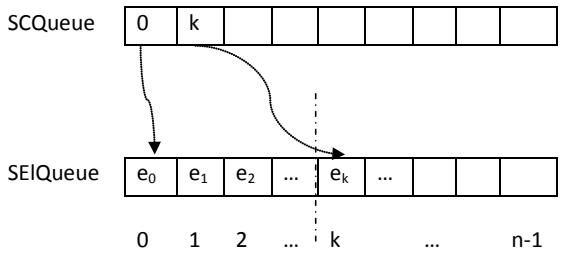

Fig. 4. `SCQueue` and `SElQueue` for the parallel version of the program

`SCQueue[]` stores the index of the first element of each small class in the parallel program (see Figure 4). It is easy for the master processor to identify not only the starting position of each small class, but also its size (the difference between the indices indicates the class size). In FSX03, the size of the small class is not important as the elements are processed sequentially; however, in the parallel implementation, the size of each small class determines the sizes of `RefStack` because the number of refinements is no more than the number of elements in the class. The size of `Refine` is the largest class index number x+1 at the previous level to ensure that `Refine` can accommodate the possible refinement from class c_x (i.e. `Refine` has index x). Thus, a possible way to decrease the amount of memory required is to allocate `Refine` and `RefStack` dynamically, and only of the necessary size (see Figure 3). This could provide a dramatic decrease on memory usage because the total memory required for `RefStack` is significantly cut down to the size of all small classes (less than string size n), and the total memory required for `Refine` is also reduced depending on the largest class index number (up to $n-1$) of the previous level.

Remark As we discussed above, it is safe to process small classes by multiple processors in parallel. However, there are several service routines used in the refinement process such as `AddToClass` or `RemoveFromClass`, etc. Their use requires mutual exclusion locking for both read and write in the parallel implementation. This is necessary because these procedures modify the shared data structures such as the classes and families and they are implemented in multiple doubly-linked lists. When these lists are accessed or modified by two processors at the same time, this could otherwise lead to corruption of these structures.

Description of the Second Approach The first approach to parallelization of FSX03 leads both to a significant speed-up and a significant increase in memory usage. In this section we describe an alternative approach to parallelizing FSX03. In a typical algorithmic "trade-off", this approach trades speed for a decrease in memory usage (see Figure 5).

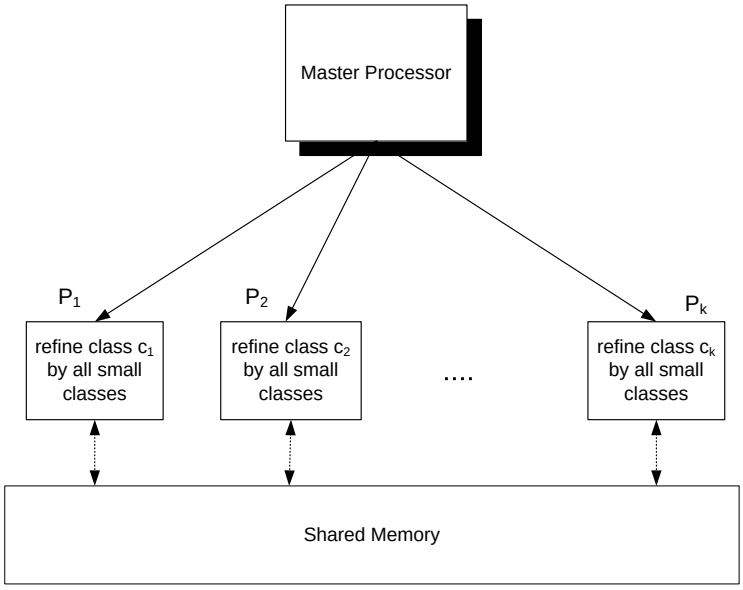

Fig. 5. Parallelization, second approach

In this approach, the master processors assigns each slave processor a class to be refined, using the "snapshot" of all small classes. The advantage of this approach is the fact that `Refine` and `RefStack` are no longer needed at all as each processor only refines one class and thus needs to "remember" only one destination class for each refinement by a small class. The disadvantage is, of course, that each of the slave processors may take almost as long as a single processor in the serial implementation of FSX03, as it must traverse the whole "snapshot". There are some potential speed-ups based on the fact, that knowing the smallest and the largest elements of the class to be refined limits the part of each small class in the "snapshot" that has to be traversed. This effect increases as the classes are getting progressively smaller.

Remark As in the first approach, the service routines used in the refinement process require mutual exclusion locking for both read and write.

4.3 Parallelization of the Extension

Extended Algorithm C/C2-K Overview In [5], we discussed implementations of three different algorithms to extend Crochemore's repetitions algorithm to compute runs. Algorithm C was the best in terms of performance (i.e. preserving the original time complexity) though it requires an extra $O(n \log n)$ space for

consolidating repetitions reported by the underlying FSX03 to runs. Its variant C2-K was introduced in order to reduce the memory requirement of C, with as little performance degradation as possible, while preserving the complexity $O(n \log n)$.

The main approach of C and C2-K is to collect all the repetitions into an array of linked lists based on their starting positions. This array acts as a list of buckets where each bucket contains a linked list of repetitions with the same starting position. After all the repetitions are collected and placed in the buckets, the algorithm traverses all the buckets and all the repetitions within each bucket and consolidates the "nearby" repetitions with the same period into runs.

In contrast to C, C2-K partially consolidates repetitions into runs by joining the repetitions from up to K buckets to the left and up to K buckets to the right of the current position during the repetitions collection process. The rationale behind this approach is lowering the memory requirements: there are fewer runs than repetitions, so storing runs rather then repetitions saves memory. C2-K guarantees that all repetitions up to period K have been fully consolidated into runs before the final sweep, while repetitions of periods $> K$ are partially consolidated.

Description The main task of the extended algorithm is to consolidate the repetitions that were previously collected in the buckets into runs. Instead of one processor carrying out the consolidation for all periods of repetitions, in the parallel version, each of the processors can be assigned to consolidate only repetitions of a given range of periods (the range could consist of as little as one period). Thus, the master processor splits up the consolidation work in terms of ranges of periods and the slave processors traverse the buckets and only consolidate the repetitions within its assigned range of periods.

Data Structure Recall (see [5]) that C2-K requires two arrays `LastRun_p` and `LastRun_s` to store the pointer and starting position, respectively, of the last run in the bucket array. For the parallelized version of C-2K, as different processors are sweeping through the buckets to consolidate the repetitions of different periods in parallel, the repetitions of any given period p are processed by one single processor. Therefore, memory locations `LastRun_p[`p`]` and `LastRun_s[`p`]` are accessed and modified by one processor at any given time. This simple observation ensures that the multiple processors can make use of `LastRun_p` and `LastRun_s` to consolidate runs in parallel without worrying about race conditions or data corruption.

Another easy observation is that at any given time, no two processors modify the same repetition in the buckets. Because every repetition has a specific period, the consolidation of repetitions with a given period is designated to one processor only. Hence, there is no extra data structure required for the parallelized implementation of C2-K. Moreover, there is no need for mutually exclusive locking of the common structures.

5 Conclusion and Further Research

We have investigated parallelization of the extended Crochemore's repetitions algorithm to compute runs within the framework of the shared memory model. The discussion has been carried out on a rather abstract level, dealing with the issues of preserving the integrity of the shared data structures and avoiding the race conditions.

Let us remark that the discussion of the parallelization of FSX03 focused mainly on the parallelization of the refinement step. However, there are other aspects that could potentially be parallelized. For example, for the computation of level 1, the input string could be partitioned into several substrings, processed by different processors in parallel and the results are pieced together by the master processor.

We identified two basic approaches:

(1) The first approach that increases the memory usage by $P \times 2n$ integers, where P is the number of available processors, but giving a large speed-up in comparison to the serial version;
   (a) and its modification that relies on dynamic memory allocation while lowering the memory demand. Of course, dynamic allocation and deallocation degrade the performance.
(2) The second approach that decreases the memory usage by $2n$ integers, but giving a small speed-up in comparison to the serial version.

We are currently working on C/C++ implementations of all three variants for a 16-core server. We will conduct experiments to compare these variants and the serial program in order to determine

(1) whether the speed-up of the first approach warrants the increased memory usage;
(2) whether the modification of the first approach using dynamic allocation brings any significant decrease in memory usage;
(3) whether the memory usage decrease of the second approach warrants the speed degradation.

In the near future, we intend to investigate all aspects of parallelization of the extended Crochemore's algorithm within the framework of the distributed memory model. Since there will not be any data structures to share, the resulting algorithm will be quite distinct in its implementation. We plan on using SHARCNET, a super-computing network, as the hardware platform for the implementation.

6 Acknowledgements

This work was supported by a grant from the Natural Sciences and Engineering Research Council of Canada.

References

1. A. Apostolico and D. Breslauer, *An Optimal O(log log n)-Time Parallel Algorithm for Detecting all Squares in a String*, SIAM J. Comput., 25, 6 (1996), 1318-1331.
2. A. Apostolico and V. E. Brimkov, *Fibonacci arrays and their two-dimensional repetitions*, Theor. Comp. Sci., 237, 2-1 (2000), 263-273.
3. M. Crochemore, *An optimal algorithm for computing the repetitions in a word*, Inform. Process. Lett., 12, 5 (1981), 244-250.
4. E. Delacourt, J.F. Myoupo, and D. Semè, *A constant time parallel detection of repetitions*, Parallel processing letters, 9, 1 (1999), 81-92.
5. F. Franek and M. Jiang, *Crochemore's repetitions algorithm revisited - computing runs*, Proceedings of Prague Stringology Conference 2009, 123-128.
6. F. Franek, W. F. Smyth, and X. Xiao, *A note on Crochemore's repetitions algorithm; a fast space-efficient approach*, Nordic Journal of Computing, 10, (2003), 21-28.
   Source code available at `www.cas.mcmaster.ca/~franek/research/croch10.cpp`
7. T. Garcia and D. Semè, *A Coarse-Grained Multicomputer algorithm for the detection of repetitions*, Inf. Process. Lett., 93, 6 (2005), 307-313.
8. K. Hirashima, H. Bannai, W. Matsubara, A. Ishino, and A. Shinohara, *Bit-parallel algorithms for computing all the runs in a string*, Proceedings of Prague Stringology Conference 2009, 203-213.
9. R. Kolpakov and G. Kucherov, *On maximal repetitions in words*, J. Discrete Algs., 1 (2000), 159-186.
10. M. G. Main, *Detecting leftmost maximal periodicities*, Discrete Applied Maths., 25, 1-2 (1989), 145-153.
11. A. Thue, *Über unendliche zeichenreihen*, Norske Vid. Selsk. Skr. I. Mat. Nat. Kl. Christiana, 7, (1906), 1-22.