A Coarse-Grained Multicomputer algorithm for the detection of repetitions

Thierry Garcia, David Semé *

LaRIA: Laboratoire de Recherche en Informatique d’Amiens, Université de Picardie Jules Verne, CURI,
5, rue du Moulin Neuf, 80000 Amiens, France

Received 17 October 2003
Available online 11 January 2005
Communicated by F. Dehne

Abstract

The paper presents a Coarse-Grained Multicomputer algorithm that solves the problem of detection of repetitions. This algorithm can be implemented in the CGM model with \( P \) processors in \( O(N^2/P) \) in time and \( O(P) \) communication steps. It is the first CGM algorithm for this problem. We present also experimental results showing that the CGM algorithm is very efficient.

Keywords: Parallel algorithms; Coarse-Grained Multicomputers; Dynamic programming; String matching

1. Introduction

This paper describes a Coarse-Grained Multicomputer solution for the detection of repetitions. Strings of symbols containing no consecutive occurrences of the same pattern have attracted the attention of researchers in diverse fields for a long time. For example, repetitions play a crucial role in many domains [3,19] such as formal language theory, term rewriting, data compression and computational molecular biology. Perhaps their first appearance dates back to the work of Thue [23], who is generally credited with the discovery of arbitrary long streams of symbols from a finite alphabet that do not contain any “square” substrings, i.e., subpatterns formed by the concatenation of some substring with itself.

In recent years, the study of such “square-free” strings has been found relevant to automata and formal language theory, algebraic coding and more generally in systems theory and combinatorics, and we shall make no attempt to refer to the existing copious literature. Suffice it to mention that papers have been devoted to the construction of arbitrarily long square-free (as well as other related repetition-constrained)
strings [18] over alphabets of fixed cardinality. In a related endeavor, the complementary notions of periodicity and overlaps of strings have been extensively investigated and still are an active research subject (see Duval [11] for an extensive bibliography).

In the framework of pattern matching [2], some classic results on string periodicity [13] have been used to develop clever techniques for the detection of assigned patterns in text-strings in time linear in the string length [7].

The problem of the efficient recognition of the occurrence of substring squares in a string $X$ stems quite naturally from the preceding remarks, and it is certainly relevant to a variety of practical applications as well [1]. $O(N^2)$ time algorithms can be quickly developed on the basis of existing pattern matching techniques and tools ($N$ is the length of the string $X$). Recently, an $O(N \log N)$ algorithm has been proposed to determine whether a given text-string over a finite alphabet contains a repetition [21]. Crochemore [5] developed an $O(N \log N)$ algorithm to determine all repetitions in a text-string $s$. Crochemore’s approach essentially relies on the well-known minimization algorithm for finite state automata [2] and exploits the theoretical bound of $N \log N$ repetitions in a string [20] as a terminating condition. Apostolico and Negro [4] presented a systolic algorithm for the detection of repetitions running in time $O(N)$ (with $4N - 3$ steps) and using $N$ processors, whereas the linear systolic array for the same problem presented in [22] requires $(5N/4) - 1$ steps on $N/4$ processors. The only known algorithm for the PRAM model can be obtained by simulating the linear systolic array on the PRAM. A constant time parallel detection of repetitions with a BSR (Broadcasting with Selective Reduction) solution has been developed by [10]. All known parallel algorithms require a work of $O(N^2)$.

In this paper we show an algorithm which solve this problem in the CGM model but this algorithm is not work optimal. Indeed, the optimality in work to solve this problem is $O(N \log N)$. Here, our algorithm has a work of $O(N^2)$ but our goal is to obtain a CGM algorithm from a systolic solution having a work of $O(N^2)$. This is a contribution to a more larger work on the adaptation of linear systolic solutions in CGM model.

In recent years several efforts have been made to define models of parallel computation that are more realistic than the classical PRAM models. In contrast of the PRAM, these new models are coarse grained, i.e., they assume that the number of processors $P$ and the size of the input $N$ of an algorithm are orders of magnitudes apart, $P \ll N$. By the precedent assumption these models map much better on existing architectures where in general the number of processors is at most some thousands and the size of the data that are to be handled goes into millions and billions.

This branch of research got its kick-off with Valiant [24] introducing the so-called Bulk Synchronous Parallel (BSP) machine, and was refined in different directions, for example, by Culler et al. [6], LogP, and Dehne et al. [9]. CGM extensively studied in [8,12,14–16].

CGM seems to be the best suited for a design of algorithms that are not too dependent on an individual architecture.

We summarize the assumptions of this model:

- all algorithms perform in so-called supersteps, that consist of one phase of interprocessor communication and one phase of local computation,
- all processors have the same size $M = O(N/P)$ of memory ($M > P$),
- the communication network between the processors can be arbitrary.

The goal when designing an algorithm in this model is to keep the individual workload, time for communication and idle time of each processor within $T/s(P)$, where $T$ is the runtime of the best sequential algorithm on the same data and $s(P)$, the speedup, is a function that should be as close to $P$ as possible. To be able to do so, it is considered as a good idea the fact of keeping the number of supersteps of such an algorithm as low as possible, preferably $O(M)$.

As a legacy from the PRAM model it is usually assumed that the number of supersteps should be poly-logarithmic in $P$, but there seems to be no real world rationale for that. In fact, algorithms that simply ensure a number of supersteps that are a function of $P$ (and not of $N$) perform quite well in practice, see Goudreau et al. [17].

The paper is organized as follows. In Section 2, we present the detection of repetition problem. The CGM solution of the problem is described in Section 3. Section 4 presents some experimental results and the conclusion ends the paper.
2. The detection of repetitions problem

2.1. Basic definitions and notations

Let $A$ be a finite alphabet and $A^*$ be the free monoid generated by $A$. Let $A^+$ be the free semigroup generated by $A$. A string $X \in A^+$ is fully specified by writing $X = x_1 \ldots x_n$, where $x_i \in A$ ($1 \leq i \leq n$).

**Definition 1.** A factor of $X$ is a substring of $X$ and its starting position in $\{1, 2, \ldots, n\}$. The notation $X[k:l]$ is used to denote the factor of $X$: $x_kx_{k+1}\ldots x_l$. A left (right) factor of $X$ is a prefix (suffix) of $X$.

**Definition 2.** A string $X \in A^+$ is primitive if setting $X = uk$ implies $u = X$ and $k = 1$.

**Definition 3.** A square in a string $X$ is any nonempty substring of $X$ in the form $uu$.

**Definition 4.** String $X$ is square-free if no substring of $X$ is a square. Equivalently, $X$ is square-free if each substring of $X$ is primitive.

**Definition 5.** A repetition in $X$ is a factor $X[k:l]$ for which there are indices $l, d$ ($k < d \leq l \leq m$) such that:

1. $X[k:l]$ is equivalent to $X[d:m]$,
2. $X[k:d-1]$ corresponds to a primitive word,
3. $X_{i+1} \neq X_{m+1}$.

**Definition 6.** $p$ is a period of a repetition $X[k:m]$ of $X$ if $x_i = x_{i+p}$ ($\forall i = k, k + 1, \ldots, |X[k:m]| - p$).

Therefore, $1 \leq p \leq |X[k:m]|/2$.

2.2. Dynamic programming approach

Let $X$ be a word of length $N$ and $p$ be a period, our goal is to detect all repetitions in $X$ for a period $p$ using Definitions 5 and 6. Table 1 gives an example of all periods of the word $X$ of length $N = 16$.

The idea is to cut this table in $P$ columns of $N/P$ elements, with $P \ll N$ and $P \neq 0$. Table 2 gives an example of this cut using $P = 4$ processors.

Fig. 2 is composed of square parts of two equal parts of $X$. Each square part is composed of a low triangular part and a high triangular part.

The dynamic programming approach is based on different systolic solutions [25]. The following section describes two sequential algorithms based on the previous definitions, which computes respectively low triangular parts and high triangular parts.

2.3. Sequential algorithms

Algorithm 1 detects all repetitions for periods $p$ (with $1 \leq p \leq N - 1$) from the first character to the
(\(N - p\))th character of the word \(X\). At the end, two tables \(I_c\) and \(L_c\) are created or modified. These tables represent respectively starting and ending positions of a repetition in the word \(X\) for a period \(p\). The different repetitions are given dynamically during the execution. So, the function \(R(I_c, p, L_c)\) is used to write the repetition starting at the position \(I_c\) with a period \(p\) of length \(L_c\).

Algorithm 2 detects all repetitions for periods \(1 \leq p \leq N\) computed from the \((N - p + 1)\)th character to the \(N\)th character of the word \(X\). At the end, two tables \(I_c\) and \(L_c\) are created or modified. These tables represent respectively starting and ending positions of a repetition in the word \(X\) for a period \(p\). The different repetitions are given in real time during the execution.

2.4. CGM algorithm

Algorithm REPET_CGM presents the CGM solution of the Detection of Repetition problem. Each processor \(num\) \((1 \leq num \leq P)\) has the \(num\)th partition of \(N/P\) elements of the input sequence \(X\). The following CGM algorithm presents the program of each processor \(num\). This CGM algorithm uses sequential Algorithms 1 and 2 described before as local computation parts. Two functions are used for communication rounds:

```plaintext
send \((X, \text{num})\): a vector \(X\) is sent to the processor \(\text{num}\),
```

```plaintext
Algorithm 1.
```
for \( (p = 1) \) to \((p = N) \)
for \((i = N - p + 1) \) to \((i = N) \)
  if \((X[i] \neq X[i + p]) \)
    \(Ic[p] = Ic[p] + Lc[p] - p + 1 \)
    \(Lc[p] = p \)
  else
    \(Lc[p] = Lc[p] + 1 \)
    if \((Lc[p] \geq 2p) \)
      \(R(Ic[p], p, Lc[p]) \)
endif
endif
endfor
endfor

Algorithm 2.

Algorithm REPET_CGM.

receive \((Y, num)\): a vector \(Y\) is received from the processor \(num\).

2.5. Communication rounds

The communication strategy is described in Fig. 1. The number of communication rounds is \(2(P - 1)\).

Proof. Each processor sends one message to the processor 0. So we have \((P - 1)\) communication rounds. After that, the message from the last processor received by the processor zero should go through all other processors. This implies \((P - 1)\) communication rounds more. Thus, the total number of communication rounds is \(2(P - 1)\). This ends the proof.

2.6. Complexity

The complexity of the CGM algorithm is \(O(N^2/P)\).

Proof. The complexity of sequential algorithms used as local computation in our CGM algorithm is \(O((N/P)^2)\). A processor must execute \(N/2\) periods. Each processor has \(N/P\) characters and compares \((N/2 \times P/N) = P/2\) characters. When a processor has finished the execution of its local computations and as the first period begins to the second character, the total complexity is \((P/2 \times 2 + 1) \times N^2/P = (P + 1) \times N^2/P\). This implies a complexity of \(O(N^2/P)\). This ends the proof.

Then, the work of our CGM solution is \(O(N^2)\) as for the systolic solution.

3. Experimental results

We have implemented these programs in C language using MPI communication library and tested them on a multiprocessor Celeron 466 MHz platform running LINUX. The communication between processors is performed through an Ethernet switch.

Tables 3 and 4 present respectively the total running times and the communication times (in seconds), for each configuration of 1, 2, 4, 8 and 16 processors.
Fig. 2. Total running times (in seconds) for different numbers of data items. The curves represent configurations of 2, 4, 8 and 16 processors, respectively.

Table 3
Total running times (in seconds) for each configuration of 1, 2, 4, 8 and 16 processors, respectively

<table>
<thead>
<tr>
<th>N</th>
<th>P = 1</th>
<th>P = 2</th>
<th>P = 4</th>
<th>P = 8</th>
<th>P = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>4.15</td>
<td>3.11</td>
<td>3.14</td>
<td>1.90</td>
<td>1.19</td>
</tr>
<tr>
<td>8192</td>
<td>16.55</td>
<td>12.41</td>
<td>13.36</td>
<td>7.68</td>
<td>5.13</td>
</tr>
<tr>
<td>16384</td>
<td>66.10</td>
<td>49.57</td>
<td>53.27</td>
<td>31.10</td>
<td>20.81</td>
</tr>
<tr>
<td>32768</td>
<td>266.09</td>
<td>199.57</td>
<td>213.87</td>
<td>124.46</td>
<td>83.85</td>
</tr>
<tr>
<td>65536</td>
<td>1086.78</td>
<td>815.09</td>
<td>868.44</td>
<td>572.94</td>
<td>335.52</td>
</tr>
<tr>
<td>131072</td>
<td>4376.22</td>
<td>3282.16</td>
<td>3512.33</td>
<td>2307.40</td>
<td>1624.47</td>
</tr>
<tr>
<td>262144</td>
<td>17536.28</td>
<td>13152.21</td>
<td>14162.64</td>
<td>10259.82</td>
<td>6569.23</td>
</tr>
<tr>
<td>524288</td>
<td>81061.04</td>
<td>60795.78</td>
<td>45294.67</td>
<td>42242.09</td>
<td>37367.71</td>
</tr>
<tr>
<td>1048576</td>
<td>324395.75</td>
<td>243296.81</td>
<td>181163.14</td>
<td>169284.73</td>
<td>150588.20</td>
</tr>
</tbody>
</table>

Table 4
Communication times (in seconds) for each configuration of 2, 4, 8 and 16 processors, respectively

<table>
<thead>
<tr>
<th>N</th>
<th>P = 2</th>
<th>P = 4</th>
<th>P = 8</th>
<th>P = 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>0.004</td>
<td>0.010</td>
<td>0.037</td>
<td>0.038</td>
</tr>
<tr>
<td>8192</td>
<td>0.007</td>
<td>0.015</td>
<td>0.037</td>
<td>0.042</td>
</tr>
<tr>
<td>16384</td>
<td>0.013</td>
<td>0.061</td>
<td>0.039</td>
<td>0.055</td>
</tr>
<tr>
<td>32768</td>
<td>0.025</td>
<td>0.088</td>
<td>0.097</td>
<td>0.085</td>
</tr>
<tr>
<td>65536</td>
<td>0.053</td>
<td>0.144</td>
<td>0.177</td>
<td>0.179</td>
</tr>
<tr>
<td>131072</td>
<td>0.104</td>
<td>0.221</td>
<td>0.508</td>
<td>0.611</td>
</tr>
<tr>
<td>262144</td>
<td>0.203</td>
<td>0.429</td>
<td>1.005</td>
<td>2.128</td>
</tr>
<tr>
<td>524288</td>
<td>0.404</td>
<td>0.847</td>
<td>2.014</td>
<td>4.367</td>
</tr>
<tr>
<td>1048576</td>
<td>0.809</td>
<td>1.686</td>
<td>4.064</td>
<td>8.852</td>
</tr>
</tbody>
</table>

The communication times correspond to send and receive $N/P$ characters by a processor to another using the communication rounds described before. Here, we have $N = 2^k$ where $k$ is an integer such that $12 \leq k \leq 20$.

Table 4 shows that communication times increase proportionally to the number of processors. But Table 3 shows that total running times, for a fixed number of data items, decrease when the number of processors increase. Then, the communication times are much more lower than the computation times. This leads that there is more important to improve local computations than communication rounds.

4. Concluding remarks

We presented a Coarse-Grained Multicomputer algorithm that solves the detection of repetitions prob-
lem. This algorithm can be implemented in the CGM model with \( P \) processors with a time complexity of \( O(N^2/P) \) and \( O(P) \) communication rounds. This is the first CGM algorithm for this problem.

The paper presents also experimental results showing that the CGM algorithm is very efficient and that it is interesting to increase the number of processors when the input sequence \( X \) is very long.

We should now implement this algorithm on another platform in order to show its portability. Our goal was to develop a CGM algorithm from a systolic solution in order to show that a linear systolic solution can be implemented in the CGM model with the same work. It will be interesting to develop another CGM algorithm for the same problem using the optimal sequential solution as local computation.

References