ELSEVIER

Pattern Recognition Letters 18 (1997) 63-72

Pattern Recognition Letters

Vectorizations of randomized matching for run-length coded strings

Kuo-Liang Chung[a,*], Wen-Ming Yan[b]

[a] *Department of Information Management, National Taiwan Institute of Technology, No. 43, Section 4, Keelung Road, Taipei, Taiwan 10672, ROC*

[b] *Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan 10764, ROC*

Received 11 April 1996; revised 21 November 1996

Abstract

Matching run-length coded strings (RLCSs) is very important in the field of pattern recognition. This paper considers the design of vectorized matching algorithms that operate directly on RLCSs. We first modify the algorithm of Karp and Rabin (1987) to design a linear-time randomized matching algorithm for RLCSs. Following this algorithm, two new and fast vectorized algorithms are presented. The first one is off-line and the second one is on-line. Some experiments are carried out on a CRAY X-MP EA/116se vector supercomputer to demonstrate the good performance of our vectorized algorithms. © 1997 Elsevier Science B.V.

Keywords: Pattern matching; Randomized algorithms; Vectorized algorithms; Run-length coded strings; Error probability; Vector supercomputer

1. Introduction

Run-length coding is well-known in the field of image processing and pattern recognition (Rosenfeld and Kak, 1982). The basic idea is to replace sequences of repeated symbols with a count of the number and that symbol. For example, the run-length coded string (RLCS) of the plain 8-connected chain code 4445556677777 is $4^3 5^3 6^2 7^5 = (4,3)(5,3)(6,2)(7,5)$ with four entries, where each entry contains one symbol and its repetitions, and the length is reduced from 13 to 8. It works well when the size of the alphabet set is small and symbols do not occur independently but are influenced by their predecessors.

Suppose a plain text (pattern) with length t (p) has been compressed into an RLCS with x (y) entries, $y \ll x$. Not only the run-length coding is a good compression method for representing strings, but designing efficient algorithms for manipulating these RCLSs is also an important research issue. Recently, an O(1)-time matching algorithm on RLCSs (Chung, 1995) is presented on a reconfigurable mesh with O(xy) processors; an O(1)-time matching algorithm for strings with variable length don't cares (Chung, 1996) is presented on

* Corresponding author. E-mail: klchung@cs.ntit.edu.tw.

a reconfigurable mesh with $O(tp)$ processors. Some fast edit distance algorithms (Bunke and Csirik, 1993; 1995) have also been developed on RLCSs.

This paper first modifies the algorithm of Karp and Rabin (1987), which is only concerned with plain text and pattern, to design a linear-time randomized matching algorithm for RLCSs. Following this algorithm, two new and fast vectorized algorithms are presented. The first one is off-line and the second one is on-line. To the best of our knowledge, this is the first time such three randomized algorithms for RLCSs are being presented in the literature. The results of this paper generalize a previous one (Chung and Yan, 1994) which is only concerned with plain text and pattern. Some experiments are carried out on a CRAY X-MP EA/116se vector supercomputer to demonstrate the good performance of the vectorized algorithms.

The remainder of this paper is organized as follows. In Section 2, the linear-time randomized matching algorithm on RLCSs is presented. Section 3 presents two fast vectorized algorithms and illustrates the related implementations on a CRAY X-MP EA/116se vector supercomputer. Some concluding remarks are addressed in Section 4.

2. Linear-time randomized matching on RLCSs

Previously, Karp and Rabin (1987) presented a linear-time randomized algorithm for pattern matching which is only concerned with plain text and pattern. In this section, we modify their algorithm to design a linear-time randomized matching algorithm on RLCSs.

We start by taking a simple example to explain the three matching conditions on RLCSs. Let the run-length coded text be

$$\begin{aligned} T &= (t^{(1)},n^{(1)})(t^{(2)},n^{(2)})(t^{(3)},n^{(3)})(t^{(4)},n^{(4)})(t^{(5)},n^{(5)})(t^{(6)},n^{(6)}) \\ &= (a,3)(c,2)(d,4)(b,3)(a,7)(b,3)(a,6) \end{aligned}$$

and the pattern be

$$P = (p^{(1)},m^{(1)})(p^{(2)},m^{(2)})(p^{(3)},m^{(3)})(p^{(4)},m^{(4)}) = (a,2)(c,2)(d,4)(b,2).$$

The matched position is in the first entry at position two since it satisfies the three matching conditions:

(1) $p^{(1)} = t^{(1)}$, $m^{(1)} \leqslant n^{(1)}$,

(2) $t^{(i)} = p^{(i)}$, $n^{(i)} = m^{(i)}$ for $2 \leqslant i \leqslant 3$ and

(3) $p^{(4)} = t^{(4)}$, $m^{(4)} \leqslant n^{(4)}$.

2.1. Fingerprinting functions

For one entry within RLCSs, we assume each symbol ranges from 0 to 255 and the repeated number of the symbol also ranges from 0 to 255 for simplicity. Let the run-length coded text with x entries and pattern with y entries be $T = (t^{(1)},n^{(1)})(t^{(2)},n^{(2)})\ldots(t^{(x)},n^{(x)})$ and $P = (p^{(1)},m^{(1)})(p^{(2)},m^{(2)})\ldots(p^{(y)},m^{(y)})$, respectively, $0 \leqslant t^{(i)},n^{(i)},p^{(j)},m^{(j)} \leqslant 255$ for $1 \leqslant i \leqslant x$ and $1 \leqslant j \leqslant y$. We first transfer each entry $(t^{(i)},n^{(i)})$ of T into an unsigned 16-bit integer by computing $t(i) = t^{(i)} * 256 + n^{(i)}$, $1 \leqslant i \leqslant x$, and each entry $(p^{(j)},m^{(j)})$ of P into an unsigned 16-bit integer by computing $p(j) = p^{(j)} * 256 + m^{(j)}$ for $1 \leqslant j \leqslant y$.

Let $l_{\text{pat1}} = p^{(1)} * 256 + m^{(1)}$, $l_{\text{pat2}} = p^{(1)} * 256 + 256$, $r_{\text{pat1}} = p^{(y)} * 256 + m^{(y)}$ and $r_{\text{pat2}} = p^{(y)} * 256 + 256$. According to the above three matching conditions and this new coding scheme, we say that a real match occurs in the ith entry at position j $(= n^{(i)} - m^{(1)} + 1)$ if the following three matching conditions hold: for $1 \leqslant i \leqslant x - y + 1$,

(1) $l_{\text{pat1}} \leqslant t(i) < l_{\text{pat2}}$,

(2) $t(i - 1 + s) = p(s)$ for $2 \leqslant s \leqslant y - 1$ and

(3) $r_{\text{pat1}} \leqslant t(i+y-1) < r_{\text{pat2}}$.

For example, suppose a (b) is assigned to 1 (2); c (d) is assigned to 3 (4), then following the above same example, we have the coded values:

$$t(1) = 256+3 = 259,\quad t(2) = 770,\quad t(3) = 1028,\quad t(4) = 515,\quad t(5) = 263,$$
$$t(6) = 515,\quad t(7) = 262,\quad p(1) = 258,\quad p(2) = 770,\quad p(3) = 1028,\quad p(4) = 514,$$
$$l_{\text{pat1}} = 258,\quad r_{\text{pat1}} = 512,\quad l_{\text{pat2}} = 514,\quad r_{\text{pat2}} = 768.$$

It is easy to verify that the three matching conditions hold, i.e.,

(1) $l_{\text{pat1}} \leqslant t(1) < l_{\text{pat2}}$,

(2) $t(2) = p(2)$, $t(3) = p(3)$ and

(3) $r_{\text{pat1}} \leqslant t(4) < r_{\text{pat2}}$.

Therefore, the pattern P matches the first entry at position two in the text T.

Following the above three matching conditions, we define two related fingerprinting functions to map run-length coded substrings into integers. Sometimes, some mapped integers are too large. As a result, overflow may occur (Kincaid and Cheney, 1996). In order to avoid overflow, employing the operations, mod p's, where p is a selected prime number and is smaller than the maximum value allowable in the machine, in the fingerprinting function, the mapped value is always smaller than p. For the text T, we define a fingerprinting function:

$$C_T(1) = (t(2)r^{y-3} + t(3)r^{y-4} + \cdots + t(y-1)) \bmod p,$$

where the practical consideration on the randomly chosen prime p will be discussed later; the value of r is naturally selected as 65536 since each $t(i)$ occupies 16 bits. For the pattern P, the fingerprinting function is defined as

$$C_P = (p(2)r^{y-3} + p(3)r^{y-4} + \cdots + p(y-1)) \bmod p.$$

We define the operations $+_p$ and $\times_p$ to be $a +_p b = (a+b) \bmod p$ and $a \times_p b = ab \bmod p$. For simplicity, suppose the time complexity spent on the operation $\times_p$ or $+_p$ is one unit. Using Horner's rule (Aho et al., 1975) and the modular property, it first takes $(4y-12)$ time to compute

$$C_P = (\ldots((p(2) \times_p r +_p p(3)) \times_p r +_p p(4))\ldots)$$

and

$$C_T(1) = (\ldots((t(2) \times_p r +_p t(3)) \times_p r +_p t(4))\ldots).$$

It then takes $(y-3)$ time to precompute $(r^{y-2} \bmod p)$ which will be used to compute $C_T(i)$ from $C_T(i-1)$. At the ith step, $2 \leqslant i \leqslant x-y+1$, we compute

$$\begin{aligned} C_T(i) &= (t(i+1)r^{(y-3)} + t(i+2)r^{(y-4)} + \cdots + t(i+y-2)) \bmod p \\ &= C_T(i-1) \times_p r +_p t(y+i-2) -_p t(i) \times_p (r^{y-2} \bmod p). \end{aligned}$$

It takes four-unit time to compute $C_T(i)$ from $C_T(i-1)$.

We say that a possible match occurs at position i when $C_T(i) = C_P$, $l_{\text{pat1}} \leqslant t(i) < l_{\text{pat2}}$ and $r_{\text{pat1}} \leqslant t(i+y-1) < r_{\text{pat2}}$. We now propose a *delay-mod* approach to speed up the computation of $C_T(i)$ from $C_T(i-1)$ without overflow. Looking at the formula

$$C_T(i-1)r + t(y+i-2) - t(i)(r^{y-2} \bmod p),$$

it is clear that the formula is less than pr and larger than $-pr$. Accordingly, $C_T(i)$ can be obtained by computing

$$(C_T(i-1)r + t(y+i-2) - t(i)(r^{y-2} \bmod p)) \bmod p,$$

where the prime satisfies $pr <$ MAX; MAX is the maximum integer allowable in the machine and $p < \mathrm{MAX}/r$. Although it takes five operations to compute $C_T(i)$ from $C_T(i-1)$, some time-saving is still gained when compared to the previous method described in the last paragraph since, in practice, computing $\times_p$ or $+_p$ is performing $\times$ or $+$ first and mod later.

2.2. The serial algorithm

Following Section 2.1, a high-level pseudo-code of the linear-time, i.e., $O(xk)$, randomized matching algorithm for RLCSs is listed below, where the constant k denotes the number of randomly chosen primes. Here, for each new chosen prime, a new fingerprinting function is computed.

Pseudo-code for serial algorithm.

```
randomly select primes $p_1, p_2, \ldots, p_k$ which are in $[2, M]$;
compute $C_T(1)$ for each chosen prime number;
compute $C_P$ for each chosen prime number;
compute Match(1);
for $i := 2$ to $x - y + 1$ do
  compute $C_T(i)$ for each chosen prime number;
  Match($i$) = 1 if it satisfies the three matching conditions for all the chosen primes;
  otherwise, Match($i$) = 0
end.
```

Let C_{Pj} denote C_P for prime p_j and $C_{Tj}(i)$ denote $C_T(i)$ for prime p_j. Then Match(i) is computed by

$$\begin{aligned}\mathrm{Match}(i) = &\ (C_{T1}(i) = C_{P1}) \wedge (C_{T2}(i) = C_{P2}) \wedge \cdots \wedge (C_{Tk}(i) = C_{Pk}) \\ &\wedge (l_{\mathrm{pat1}} \leqslant t(i) < l_{\mathrm{pat2}}) \wedge (r_{\mathrm{pat1}} \leqslant t(i+y-1) < r_{\mathrm{pat2}}),\end{aligned}$$

where "$\wedge$" denotes a logical AND operation.

The detailed randomized matching algorithm for RLCSs is shown below.

Algorithm String-Matching.

```
Input: $t(1:x), p(1:y)$: arrays for text and pattern, respectively;
Output: Match$(1 : x-y+1)$: boolean array;
begin
for $j := 1$ to $k$ do
  $p_j :=$ randomly chosen prime in $[2, M]$; /* $M$ is determined in Section 2.3 */
for $j := 1$ to $k$ do /* $k$ denotes the number of chosen primes */
  begin
  /* use Horner's rule to compute $C_{Tj}(1)$ which denotes $C_T(1)$ for prime $p_j$ */
  $C_{Tj}(1) = (t(2)r^{y-3} + t(3)r^{y-4} + \cdots + t(y-1)) \bmod p_j$;
  /* use Horner's rule to compute $C_{Pj}$ which denotes $C_P$ for prime $p_j$ */
  $C_{Pj} = (p(2)r^{y-3} + p(3)r^{y-4} + \cdots + p(y-4)) \bmod p_j$;
  end;
/* check the three matching conditions for $p_j$, $1 \leqslant j \leqslant k$ */
Match(1) := $(C_{T1}(1) = C_{P1}) \wedge (C_{T2}(1) = C_{P2}) \wedge \cdots \wedge (C_{Tk}(1) = C_{Pk})$
            $\wedge (l_{\mathrm{pat1}} \leqslant t(1) < l_{\mathrm{pat2}}) \wedge (r_{\mathrm{pat1}} \leqslant t(y) < r_{\mathrm{pat2}})$;
for $i := 2$ to $x - y + 1$ do
  begin
```

```
    for $j$ := 1 to $k$ do
      begin /* delay-mod computation */
      $C_{Tj}(i) := (C_{Tj}(i-1)r + t(y+i-2) - t(i)(r^{y-2} \bmod p_j)) \bmod p_j$;
      end;
    Match($i$) := $(C_{T1}(i) = C_{P1}) \wedge (C_{T2}(i) = C_{P2}) \wedge \cdots \wedge (C_{Tk}(i) = C_{Pk})$
                  $\wedge (l_{\text{pat1}} \leqslant t(i) < l_{\text{pat2}}) \wedge (r_{\text{pat1}} \leqslant t(i+y-1) < r_{\text{pat2}})$;
  end;
end.
```

2.3. Error probability

According to the pigeonhole principle (Grimaldi, 1994), a false match may occur if, for some i, the algorithm determines that $C_T(i) = C_P$ but $t(i-1+s) \neq p(s)$ for $2 \leqslant s \leqslant y-1$. The probability of such an error can be reduced to a truly negligible level if the algorithm is repeated k times and it always reports that a possible match has occurred.

Following the similar proving technique in (Karp and Rabin, 1987), the probability that a false match occurs is $\leqslant (\pi(8(y-2)(x-y-1))/\pi(M))^k$, where k is the number of repetitions for running the proposed randomized algorithm; $y-2$ is the true length of the pattern in the coding scheme; $x-y-1$ is the number of possible matchings; $\pi(u)$ denotes the number of primes $\leqslant u$ (Rosser and Schoenfeld, 1962). For saving space, we omit the detailed proof. For each chosen prime, in order to guarantee the error probability being much less than 1, we select M to be much larger than $8(y-2)(x-y+1)$. The chosen prime p must belong to $[2, M]$, M is less than MAX$/r$ since we already know that p is less than MAX$/r$ in order to avoid overflow.

3. Vectorizations

In this section, we transform the serial algorithm into a vector one to fit the features (e.g., vector length and proper memory stride) of vector supercomputers (Hwang and Briggs, 1984). Two fast vectorized matching algorithms for RLCSs are presented. One is off-line and the other is on-line.

3.1. Off-line vectorized algorithm

In this subsection, we present the off-line vectorized matching algorithm for RLCSs. Recall that there are $x-y+1$ possible matchings to be examined. Our main concept is transforming the one-dimensional operation space described in Section 2.2 into a two-dimensional one. It brings out the design of vectorized operations. That is, one vector operation is used to replace one scalar operation described in Section 2.2. In order to design the vectorized algorithm with vector length $v = 64$ and vector stride $= q$, we define two vectors as shown below:

$$\boldsymbol{T}_i = \begin{bmatrix} t(i) \\ t(q+i) \\ \vdots \\ t((v-1)q+i) \end{bmatrix} \quad \text{and} \quad \boldsymbol{C}_i = \begin{bmatrix} C_T(i) \\ C_T(q+i) \\ \vdots \\ C_T((v-1)q+i) \end{bmatrix}.$$

It is easy to derive that

$$\boldsymbol{C}_i = (\boldsymbol{T}_{i+1} r^{y-3} + \boldsymbol{T}_{i+2} r^{y-4} + \cdots + \boldsymbol{T}_{i+y-2}) \bmod p$$

for $1 \leqslant i \leqslant q$. After computing all $\boldsymbol{C}_i$'s for $1 \leqslant i \leqslant q$, we can compute all $C_{T(i)}$'s for $1 \leqslant i \leqslant vq$.

Using Horner's rule and the modular property, it first takes $(2y-6)$ vector operations with vector length v to compute

$$C_1 = (\dots((T_2 \times_p r +_p T_3) \times_p r +_p T_4))\dots).$$

Based on the delay-mod approach, it then takes five vector operations with the same vector length to compute C_i from C_{i-1}, $2 \leqslant i \leqslant q$, by calculating

$$C_i = C_{i-1}r + T_{y+i-2} - T_i(r^{y-2} \bmod p))\bmod p.$$

Let

$$M_i = \begin{bmatrix} \text{Match}(i) \\ \text{Match}(q+i) \\ \vdots \\ \text{Match}((v-1)q+i) \end{bmatrix},$$

for $1 \leqslant i \leqslant x-y+1$. Then the high-level pseudo-code of the off-line vectorized algorithm is listed below.

Pseudo-code for off-line vectorized algorithm.

```
randomly select primes $p_1, p_2, \dots, p_k$ which are in $[2, M]$;
compute $C_1$ for each chosen prime number;
compute $C_P$ for each chosen prime number;
compute $M_1$;
for $i := 2$ to $q$ do
  compute $C_i$ for each chosen prime number;
  compute $M_i$;
end.
```

Let $C_i^{(j)}$ denote $C_T(i)$ for prime p_j and let the logical operation "op" between a vector and a scalar be defined by

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_v \end{bmatrix} \text{ op } c = \begin{bmatrix} x_1 \text{ op } c \\ x_2 \text{ op } c \\ \vdots \\ x_v \text{ op } c \end{bmatrix}.$$

Then M_i is computed by

$$M_i = (C_i^{(1)} = C_{P1}) \wedge (C_i^{(2)} = C_{P2}) \wedge \cdots \wedge (C_i^{(k)} = C_{Pk}) \wedge (l_{\text{pat1}} \leqslant T_i < l_{\text{pat2}}) \wedge (r_{\text{pat1}} \leqslant T_{i+y-1} < r_{\text{pat2}}).$$

The detailed off-line vectorized algorithm is shown below.

```
**Algorithm Off-Line.**
**Input:** $t(1:x)$, $p(1:y)$: arrays;
**Output:** Match$(1:x-y+1)$: boolean array;
**begin**
**for** $j := 1$ to $k$ **do** $p_j :=$ randomly chosen prime in $[2, M]$;
```

```
/* assign zeroes to the extra remaining $t(i)$'s */
for $i := x+1$ to $v \times q + y - 1$ do $t(i) := 0$;
for $j := 1$ to $k$ do
  begin
  /* use Horner's rule to compute $C_1^{(j)}$ which denotes $C_1$ for prime $p_j$ */
  $C_1^{(j)} := (T_2 r^{y-3} + T_3 r^{y-4} + \cdots + T_{y-1}) \bmod p_j$;
  /* use Horner's rule to compute $C_{Pj}$ which denotes $C_P$ for prime $p_j$ */
  $C_{Pj} = p(2) r^{y-3} + p(3) r^{y-4} + \cdots + p(y-1)$;
end;
/* check the three matching conditions for $p_j$, $1 \leqslant j \leqslant k$ */
$M_1 := (C_1^{(1)} = C_{P1}) \wedge (C_1^{(2)} = C_{P2}) \wedge \cdots \wedge (C_1^{(k)} = C_{Pk}) \wedge (l_{\text{pat1}} \leqslant T_1 < l_{\text{pat2}}) \wedge (r_{\text{pat1}} \leqslant T_y < r_{\text{pat2}})$;
for $i := 2$ to $q$ do
  begin
  for $j := 1$ to $k$ do
    begin
    /* $C_i^{(j)}$ denotes $C_i$ for prime $p_j$ */
    $C_i^{(j)} := (C_{i-1}^{(j)} r + T_{y+i-2} - T_i (r^{y-2} \bmod p_j)) \bmod p_j$;
    end;
  $M_i := (C_i^{(1)} = C_{P1}) \wedge (C_i^{(2)} = C_{P2}) \wedge \cdots \wedge (C_i^{(k)} = C_{Pk}) \wedge (l_{\text{pat1}} \leqslant T_i < l_{\text{pat2}}) \wedge (r_{\text{pat1}} \leqslant T_{i+y-1} < r_{\text{pat2}})$;
  end;
end.
```

It can be verified that the number of vector operations required in the above vectorized algorithm is $O(kx/v)$, where v is the vector length.

3.2. On-line vectorized algorithm

Modifying the off-line algorithm of Section 3.1, we present the fast on-line vectorized string-matching algorithm for RLCSs whose input data is read in an on-line manner. The main concept of the on-line vectorized algorithm is partitioning the two-dimensional operation space, i.e., matrix form, into some disjoint submatrices. Successively, we process each submatrix in a highly vectorized way.

We define two vectors as shown below:

$$T_i^{(b)} = \begin{bmatrix} t(i+bqv) \\ t(q+i+bqv) \\ \vdots \\ t((v-1)q+i+bqv) \end{bmatrix} \quad \text{and} \quad C_i^{(b)} = \begin{bmatrix} C_T(i+bqv) \\ C_T(q+i+bqv) \\ \vdots \\ C_T((v-1)q+i+bqv) \end{bmatrix}.$$

We want to compute

$$C_i^{(b)} = (T_{i+1}^{(b)} r^{y-3} + T_{i+2}^{(b)} r^{y-4} + \cdots + T_{i+y-2}^{(b)}) \bmod p$$

for $1 \leqslant i \leqslant q$ and $0 \leqslant b \leqslant (d-1)$. Using Horner's rule and modular properties, it takes $(2y-6)$ vector operations with vector length v to compute

$$C_1^{(b)} = (\ldots((T_2^{(b)} \times_p r +_p T_3^{(b)}) \times_p r +_p T_4^{(b)}))\ldots).$$

Using the delay-mod approach, it then takes five vector operations with the same vector length to compute $C_i^{(b)}$ from $C_{i-1}^{(b)}$, $2 \leqslant i \leqslant q$, by calculating $C_i^{(b)} = (C_{i-1}^{(b)} r + T_{y+i-2}^{(b)} - T_i^{(b)} r^{y-2}) \bmod p$. Let

$$M_i^{(b)} = \begin{pmatrix} M(i+bqv) \\ M(q+i+bqv) \\ \vdots \\ M((v-1)q+i+bqv) \end{pmatrix}.$$

Then the high-level pseudo-code of the on-line vectorized algorithm is listed below.

Pseudo-code for on-line vectorized algorithm.

```
randomly select primes $p_1, p_2, \ldots, p_k$ which are in $[2, M]$;
compute $C_P$ for each chosen prime number;
**for** $b := 0$ to $d-1$ **do**
  compute $C_1^{(b)}$ for each chosen prime number;
  compute $M_1^{(b)}$;
  **for** $i := 2$ to $q$ **do**
    compute $C_i^{(b)}$ for each chosen prime number;
    compute $M_i^{(b)}$;
  **end**;
**end**.
```

The detailed on-line vectorized algorithm is shown below.

Algorithm On-Line.

```
**Input:** $t(1:x)$, $p(1:y)$: arrays;
**Output:** Match$(1 : x-y+1)$: boolean array;
**begin**
**for** $j := 1$ to $k$ **do** $p_j :=$ randomly chosen prime in $[2, M]$;
/* assign zeroes to the extra remaining $t(i)$'s */
**for** $i := x+1$ to $v \times q \times d + y - 1$ **do** $t(i) := 0$;
**for** $j := 1$ to $k$ **do** /* use Horner's rule */
  $C_{Pj} = (p(2)r^{y-3} + p(3)r^{y-4} + \cdots + p(y-1)) \bmod p_j$;
**for** $b := 0$ to $d-1$ **do**
  **begin**
  **for** $j := 1$ to $k$ **do**
  /* use Horner's rule; $C_1^{(j)} = C_1^{(b)} \bmod p_j$ */
  $C_1^{(j)} := (T_2^{(b)} r^{y-3} + T_3^{(b)} r^{y-4} + \cdots + T_{y-1}^{(b)}) \bmod p_j$;
  /* check the three matching conditions for $p_j$, $1 \leqslant j \leqslant k$ */
  $M_1^{(b)} := (C_1^{(1)} = C_{P1}) \wedge (C_1^{(2)} = C_{P2}) \wedge \cdots \wedge (C_1^{(k)} = C_{Pk})$
        $\wedge (l_{\text{pat1}} \leqslant T_1^{(b)} < l_{\text{pat2}}) \wedge (r_{\text{pat1}} \leqslant T_y^{(b)} < r_{\text{pat2}})$;
  **for** $i := 2$ to $q$ **do**
    **begin**
    **for** $j := 1$ to $k$ **do**
      **begin**
      /* $C_i^{(j)} = C_i^{(b)} \bmod p_j$ */
```

Table 1
Performance comparison of three algorithms on CRAY X-MP EA/116se supercomputer

x	Serial time_1	Off-line time_2	On-line time_3	$\text{time}_1/\text{time}_2$ R_{12}	$\text{time}_1/\text{time}_3$ R_{13}	$\text{time}_2/\text{time}_3$ R_{23}
10000	20.895	4.587	5.580	4.5553	3.7446	0.8220
20000	41.798	8.620	9.737	4.8490	4.2927	0.8853
30000	62.703	13.097	15.312	4.7876	4.0950	0.8553
40000	83.604	16.345	19.472	5.1150	4.2935	0.8394
50000	104.509	21.477	25.043	4.8661	4.1732	0.8576
60000	125.409	25.603	29.207	4.8982	4.2938	0.8766
70000	146.317	30.548	34.778	4.7897	4.2072	0.8784
80000	167.220	34.173	38.935	4.8933	4.2949	0.8777
90000	188.126	38.502	44.509	4.8861	4.2267	0.8650

```
            $C_i^{(j)} := (C_{i-1}^{(j)} r + T_{y+i-2}^{(b)} - T_i^{(b)}(r^{y-2} \bmod p_j)) \bmod p_j$;
         end;
         $M_i^{(b)} := (C_i^{(1)} = C_{P1}) \wedge (C_i^{(2)} = C_{P2}) \wedge \cdots \wedge (C_i^{(k)} = C_{Pk})$
                 $\wedge\ (l_{\text{pat1}} \leqslant T_i^{(b)} < l_{\text{pat2}}) \wedge (r_{\text{pat1}} \leqslant T_{i+y-1}^{(b)} < r_{\text{pat2}})$;
      end;
   end;
end.
```

It can be verified that the number of vector operations required in this algorithm is $O(kx/v)$.

3.3. Experimentations

In this section, we implement our serial and two vectorized algorithms in CRAY Fortran 77 on the CRAY X-MP EA/116se supercomputer. The used supercomputer has a register-to-register architecture without cache memory and has one vector processor which contains eight 64-bit vector registers. That is, the vector length is 64. Memory is divided into 16 banks and each bank contains 1M 64-bit words. Each bank requires 14 cycle time (one cycle time needs 8.5 nanoseconds) before it is ready for another request.

As to the off-line vectorized algorithm described in Section 3.1, in order to avoid the memory conflict in CRAY X-MP (Levesque and Williamson, 1989), we set q to be an as small as possible odd number such that $vq \geqslant x - y + 1$. The extra remaining $t(j)$'s for $x + 1 \leqslant j \leqslant vq + y - 1$ are set to zero. As to the on-line vectorized algorithm described in Section 3.2, in order to avoid the memory conflict, we set q to be an odd number, say $q = 4y + 1$, where q (v) is the number of columns (rows) in each submatrix. Similarly, the extra remaining $t(j)$'s for $x + 1 \leqslant j \leqslant vqd + y - 1$ are assigned to zeroes, where $d = \lceil (x - y + 1)/vq \rceil$.

The length of the text T is specified to be 10000, 20000, 30000, ..., and 90000, respectively, and for convenience, the length of the pattern P is specified to be 11. To save space, we omit the detailed source codes. The operating system used here is UNICOS 6.1.6 and the compiler is called CF77 (Supercomputer Programming (I), 1991). Table 1 shows the performance of these three algorithms on CRAY X-MP EA/116se.

In Table 1, the symbol x denotes the length of T; the length of the pattern is $y = 11$; the symbols time_1, time_2 and time_3 in terms of milliseconds, represent the time spent in the serial algorithm, the off-line one and the on-line one, respectively. In the same table, the value of R_{ij} denotes the ratio of time_i over time_j for $1 \leqslant i \neq j \leqslant 3$. The plot of time comparison is shown in Fig. 1.

It is observed from Fig. 1 that both our off-line and on-line vectorized algorithms are about four times faster than the proposed serial one.

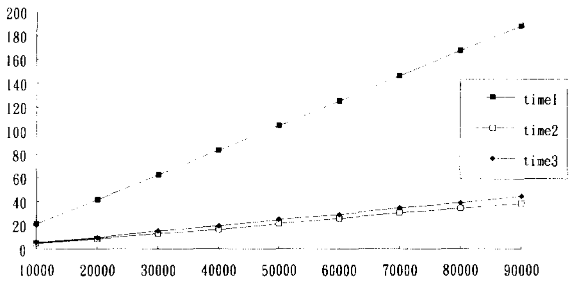

Fig. 1. The plot of time comparison of three algorithms on CRAY X-MP EA/116se supercomputer.

4. Conclusions

The significance of matching on RLCSs is its popular use in the fields of information retrieval and pattern analysis. Our main contribution is to present one serial randomized algorithm and two new and fast vectorized algorithms for matching on RLCSs. Experiments for our three algorithms have been carried out on a CRAY X-MP EA/116se supercomputer. Our results can not only be used for the applications of matching multiple patterns (Fan and Su, 1993) and the calculation of moving sum (Chung and Yan), but also generalize a previous algorithm (Chung and Yan, 1994) which is only concerned with plain text and pattern.

Acknowledgements

The authors thank the two referees and Prof. E. Backer for their constructive comments and criticism that improve the presentation and quality of this paper. This research was supported in part by the National Science Council of R.O.C. under contracts NSC86-2213-E011-010 and NCHC86-08-015. We also thank the Computer Center of the National Taiwan University and the National Center for High-Performance Computing in Taiwan for their support.

References

Aho, A., J.E. Hopcroft and J.D. Ullman (1975). *The Design and Analysis of Algorithms*. Addison-Wesley, Reading, MA, 21, 438-439.

Bunke, H. and J. Csirik (1993). An algorithm for matching run-length coded strings. *Computing* 50, 297-314.

Bunke, H. and J. Csirik (1995). An improved algorithm for computing the edit distance of run-length coded strings. *Inform. Process. Lett.* 54, 93-96.

Chung, K.L. (1995). Fast string matching algorithms for run-length coded strings. *Computing* 54 (2), 119-126.

Chung, K.L. (1996). O(1)-time parallel string-matching algorithm with VLDCs. *Pattern Recognition Letters* 17 (5), 475-479.

Chung, K.L. and W.M. Yan (1994). Vectorized computations for string matchings. Research Report, Department of Information Management, National Taiwan Inst. of Technology, June 1994.

Chung, K.L. and W.M. Yan (1995). Fast vectorization for Calculating a moving sum. *IEEE Trans. Comput.* 44 (11), 1335-1337.

Fan, J.J. and K.Y. Su (1993). An efficient algorithm for matching multiple patterns. *IEEE Trans. Knowledge and Data Engrg.* 5, 339-351.

Grimaldi, R.P. (1994). *Discrete and Combinatorial Mathematics: An Applied Introduction*, 3rd edition. Addison-Wesley, Reading, MA, 275-278.

Hwang, K. and F. Briggs (1984). *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, Chapter 4.

Karp, R.M. and M.O. Rabin (1987). Efficient randomized pattern-matching algorithms. *IBM J. Res. Develop.* 31, 249-260.

Kincaid, D. and W. Cheney (1996). *Numerical Analysis*, 2nd edition. Brooks/Cole, New York, 47.

Levesque, J.M. and J.W. Williamson (1989). *A Guidebook to Fortran on Supercomputers*. Academic Press, New York.

Rosenfeld, A. and A. Kak (1982). *Digital Picture Processing*. Academic Press, New York.

Rosser, J.B. and J. Schoenfeld (1962). Approximate formulas for some functions of prime numbers. *Illinois J. Math.* 6, 64-94.

Supercomputer Programming (I): Advanced Fortran: Architecture, Vectorization, and Parallel Computing. Working manual for CRAY X-MP EA/116se (1991).