$O(k)$ Parallel Algorithms for Approximate String Matching

Alden H. Wright   Yi Jiang

Department of Computer Science
University of Montana
Missoula, MT 59812

May 25, 1993

Approximate String Matching (proposed running head)

Professor Alden H. Wright
Dept of Computer Science
Univ. of Montana
Missoula, MT 59812

ABSTRACT

Given a text string T of length n, a shorter pattern string A of length m, and an integer k, an simple straightforward $O(k)$ parallel algorithm for finding all occurrences of the pattern string in the text string with at most k differences (as defined by edit distance) is presented. The algorithm uses the priority CRCW-PRAM model of computation and $(n - m + k + 2) \times m = O(n \times m)$ processors.
Over recent decades, *the k-differences string matching problem* received much attention in the research literature, due to its importance in molecular biology, text retrieval, and other types of applications. Based on a variation of Ukkonen's algorithm, this paper presents several different ways of achieving a time complexity of $O(k)$ for solving this problem with different CRCW-PRAM assumptions.

Key words. approximate string matching, edit distance, PRAM, CRCW-PRAM

1 Introduction

1.1 The problem

In many branches of scientific research, it is often necessary to search a large text database for approximate occurrences of an interesting pattern. This is especially important in molecular biology and evolution [Lander91], and information retrieval in general.

The *edit distance* between strings A and B, $d(A,B)$ is defined as the minimum number of edit steps in converting A to B using the following 3 kinds of edit steps:

(1) Delete an element from A.
(2) Insert an element to A.
(3) Replace an element of A with another element.

The edit distance can be generalized by putting different costs on the operations or on the elements of the alphabet. In particular, the longest common subsequence problem reduces to an edit distance problem where only the deletion and insertion operations are used.

The *k-differences approximate string matching problem* is to find all occurrences of substrings of a text string T whose edit distance from a pattern string A is less than k.

For example, let $T = abcdefgh$ and let $A = bdxfyg$. Then A occurs starting at the second position of the text with 3 differences, as illustrated by the following alignment.

$$\begin{array}{cccccccc} a & b & c & d & e & f & & g & h \\ & b & & d & x & f & y & g & \end{array}$$

The substring $bcdefg$ of T can be transformed into A by deleting c, replacing e with x, and inserting y after f.

In this paper, several different ways of achieving a parallel time complexity of $O(k)$ for solving the *k-differences string-matching problem* with different CRCW-PRAM assumptions are presented.

We use A_i to denote the i^{th} element of string A, and A_i^j to denote substring $A_iA_{i+1}\dots A_j$, and we use $d(A,B)$ to denote the edit distance between two strings A and B.

1.2 Previous work

Historically, the development of algorithms to compute edit distance between two strings began in biology with the discovery of a dynamic programming algorithm by Needleman and Wunsch [Needleman70]. and then given independently in at least eight additional papers. Sankoff and Kruskal's book [Sankoff83] references these and contains descriptions of various dynamic programming algorithms. The basic dynamic programming algorithm requires time and space which is $\Theta(mn)$ where m and n are the lengths of the two strings. Ukkonen [Ukkonen83] gave an alternative algorithm whose time is $\Theta(k \cdot min(m,n))$ where k is the edit distance between the two strings.

Sellers [Sellers79], [Sellers80] adapted the edit-distance dynamic programming algorithm to the k-differences problem. (See also [Erickson83].) Included are theorems that justify the approach used in this paper. Ukkonen [Ukkonen85a] gave an algorithm for the k-approximate string matching problem which used a preprocessing phase and a search phase. The preprocessing phase runs in time $O(m \cdot |\Sigma| \cdot K)$ where $K = min(3^m, 2^k \cdot |\Sigma|^k \cdot m^{k+1})$, and the search phase runs in time $O(n)$. Due to the exponential dependence of the preprocessing time on the pattern length, the algorithm is only practical if the pattern is very short.

Landau and Vishkin [Landau86] gave an algorithm for the k-mismatches problem which runs in $O(k(n + mlog(m))))$ time. They gave another algorithm [Landau88] for the k-differences problem that runs in $O(m + nk^2)$ assuming a fixed alphabet size. A later paper [Landau89] gave a serial algorithm that runs in $O(nk)$ and a parallel (CRCW-PRAM) algorithm using n processors that runs in $O(log(m) + k)$ time.

Galil and Park [Galil90] point out that the Landau and Vishkin algorithms of [Landau88] and [Landau89] make use of suffix trees and a lowest common ancestor algorithm, which makes them complex and less suitable for practical use. The fastest Landau and Vishkin serial algorithm for the k-difference problem that does not make use of a suffix tree is given in [Landau88], and runs in time $O(k^2n + m^2)$. Galil and Park adapted one of Ukkonen's algorithms [Ukkonen85b] to the approximate string matching problem. While this algorithm has worst-case complexity $O(mn)$, it has complexity

$O(kn)$ on random strings, and is very fast in practice. Then they give a new algorithm which achieves worst-case time complexity $O(kn + m^2)$, but is not usually as fast in practice as the previously mentioned algorithm.

Wu and Manber [Wu92] give a bit-parallel algorithm for the k-differences problem which is based on the work of Baeza-Yates and Gonnet [Baeza-Yates92]. This algorithm has a preprocessing phase which requires $O(m|\Sigma| + k\lceil\frac{m}{w}\rceil)$ steps, where w is the word-length of the computer. Then the search phase runs in $O(nk\lceil\frac{m}{w}\rceil)$ steps. While this algorithm breaks no new ground in asymptotic complexity, it is very fast in practice for short patterns (especially when the pattern is shorter than the word length). It also can be extended to support patterns that include regular expressions.

1.3 Relation of this paper to previous work

The algorithm of this paper improves on the $O(log(m) + k)$ time complexity bound of Landau and Vishkin [Landau89]. The algorithm is also far simpler than that of Landau and Vishkin, and does not use a suffix tree. However, their algorithm uses n processors, and ours uses $O(mn)$ processors. In addition, our algorithm depends on specific assumptions about the implementation of concurrent write operations in a CRCW-PRAM model of computation.

1.4 Models of computation

This paper uses the concurrent-read concurrent-write (CRCW) parallel random access machine (PRAM) model of computation. A PRAM employs p synchronous processors, all having access to a common memory. A CRCW PRAM allows simultaneous access to the common memory on both read and write operations. When multiple processors write to the same memory location, the effect is not well defined without additional elaboration.

Several assumptions have been used in the literature:

a. Arbitrary. The value written is arbitrarily chosen among the writers.

b. Priority. Store the value written by the lowest-indexed processor.

c. Combination. Combine the written values by some associative and commutative operation such as summation or maximum.

d. Common−value. When multiple processors write to the same location, they must write the same thing.

Different assumptions affect the complexity of our algorithm in different ways. References on PRAM models include [Cormen90], [Almasi89].

1.5 The basic dynamic programming algorithm

The basic version of dynamic programming method used to solve the approximate string matching problem is briefly reviewed bellow (See [Sellers79] and [Galil90]).

For simplicity, we assign a cost of 1 to each of the three edit steps.

Suppose we have a pattern string A and a text string T of lengths m and n respectively, and an integer k ($k \leq m \leq n$). Define $D[i,j]$ ($0 \leq i \leq m$, $0 \leq j \leq n$) to be the minimum edit distance from A_1^i to any substring of T ending at T_j.

One can observe that for $i \leq m$ and $j \leq n$, the edit distance between A_1^i and T_h^j for some h $(1 \leq h \leq j)$ is either

(i) $d(A_1^{i-1}, T_h^{j-1}) + d(A_i, T_j)$, or
(ii) $d(A_1^i, T_h^{j-1}) + 1$ (i.e., insert T_j), or
(iii) $d(A_1^{i-1}, T_h^j) + 1$ (i.e., delete A_i).

Thus,

$$D[i,j] = \min(D[i-1,j] + 1, D[i,j-1] + 1, D[i-1,j-1] + c_{i,j})$$

where $c_{i,j}$ denotes $d(A_i, T_j)$.

Initially, $D[i,0] = i$ for $0 \leq i \leq m$ because A_1^i differs from the empty text by i insertion steps. $D[0,j] = 0$ for $0 \leq j \leq n$ because the empty pattern occurs any where in T. $D[m,j] \leq k$ if and only if A occurs at position T_j with an edit distance at most k.

Thus, the essence of the dynamic programming method for this problem is to iteratively fill out the matrix $D[0..m, 0..n]$, either row-by-row or column-by-column. The last row of D matrix, $D[m,j]$ for $1 \leq j \leq n$, is the output of this algorithm. More specifically, if a

$D[m, j]$ is less or equal to k, then, there is an approximate occurrence of pattern A in T ending at position j, with an edit distance less than or equal to k. (See [Sellers79] and [Sellers80].)

1.6 Ukkonen's algorithm

We give another serial algorithm. The basic idea is due from Ukkonen's algorithm for the edit-distance problem [Ukkonen83]. Landau and Vishkin [Landau89] adapted the algorithm to the k-differences problem, and it is their version that we present below. Galil and Park [Galil90] give a slightly different version as Algorithm MN2.

Let *diagonal–d* of the matrix D consist of all $D[i, l]$'s such that $l - i = d$. Let e denote the number of differences, and let $L[d, e]$ denote the largest row i such that $D[i, l] = e$ and $D[i, l]$ is on diagonal d. By definition, e is the minimum edit distance between $A_1^{L[d,e]}$ and any substring of T which ends at $T_{L[d,e]+d}$. The definition of $L[d, e]$ also implies that $A_{L[d,e]+1} \neq T_{L[d,e]+d+1}$ for $L[d, e] < m$.

For each diagonal d, we are interested in finding $L[d, e]$ for $e = 0, \ldots, k$. If $L[d, e] = m$ for some e, then there is an occurrence of pattern A in T with at most k differences that ends at T_{d+m}.

In computing $L[d, e]$'s, we use induction on e. For a given d and e, suppose $L[y, x]$ has been computed for all $x < e$ and all diagonals y, and that $L[d, e] = i$, i.e., i is the largest row such that $D[i, l] = e$, and $D[i, l]$ is on diagonal d. By *Algorithm* 1, there are two possibilities for $D[i, l]$ in getting its value e:

(i) $D[i-1, l-1] = e-1$ (where $D[i-1, l-1]$ is the predecessor of $D[i, l]$ on diagonal d) and $A_i \neq T_l$. Or, either $D[i, l-1]$ or $D[i-1, l]$ has a value of $e - 1$.

(ii) $D[i - 1, l - 1] = e$ and $A_i = T_l$.

This implies that we may begin from $D[i, l]$ and go upward along diagonal d by possibility (ii) until we get a possibility (i). Putting it in a reverse way, we can compute $L[d, e]$ by first locating the row of diagonal d where possibility (i) occurs, and then increasing the row number as long as possibility (ii) occurs.

These observations leads to the following algorithm:

Algorithm 1:

Initialization:

  for $0 \leq d \leq n$ do $L[d,e] \leftarrow -1$

  for $-(k+1) \leq d \leq -1$ do $L[d, \mid d \mid -1] \leftarrow \mid d \mid -1$, $L[d, \mid d \mid -2] \leftarrow \mid d \mid -2$

  for $-1 \leq e \leq k$ do $L[n+1,e] \leftarrow -1$

Main loop:

  for $e = 0$ to k do

    for $d = -e$ to n do

1    $row \leftarrow \max[\ (L[d,e-1]+1), (L[d-1,e-1]), (L[d+1,e-1]+1)\ ]$

    $row \leftarrow \min(row, m)$

2   while $row < m$ and $row + d < n$ and $A_{row+1} = T_{row+1+d}$ do

      $row \leftarrow row + 1$

3    $L[d,e] \leftarrow row$

4   if $L[d,e] = m$ then

      print "There is an occurrence ending at T_{d+m}"

As can be seen, this algorithm computes $L[d,e]$'s along $n+k+1$ diagonals. Furthermore, since in the inner loop, the variable row can take at most m different values, this algorithm has a time complexity of $O(mn)$ (in practice, this algorithm is faster than the basic algorithm, due to the fact it never computes values for D cells which have cost greater than k).

It is not difficult to recover the sequence of edit steps from the L matrix. See [Ukkonen85b] for details.

1.7 Landau and Vishkin's parallel algorithm

Landau and Vishkin's [Landau89] parallel algorithm is a parallelization of algorithm 1 which runs in $O(log(m)+k)$ using n processors.

First, observe that the second for-loop in algorithm 1 can be done in parallel by $O(n)$ processors. Then, observe that in the inner loop of algorithm 1, statement 2 is a while loop which carries the task of finding $L[d,e]$. The key idea in Landau and Vishkin's algorithm is to use a suffix tree data structure to find $L[d,e]$ in $O(1)$ time. The suffix tree is to be built beforehand.

Since the time needed to construct the suffix tree is $O(\log\ n)$ [Apostolico88], the total time is $O(k+ \log\ n)$ [Landau89].

Finally, using the divide-and-conquer method, one can split the original problem into $\lceil n/m \rceil$ smaller problems each of size $O(m)$, and solve them in parallel using the above algorithm and $\lceil n/m \rceil$ groups of processors, thus achieving a time complexity of $O(k+\log m)$[Landau89].

2 $O(k)$ time with $O(m \times n)$ processors

Let's consider again algorithm 1 (Ukkonen's algorithm).

Since the inner for-loop can be done in parallel, if we can execute statement 2 (while-loop) in $O(1)$, then, the main loop can be done in $O(k)$. This is the motivation for the use of the suffix tree in Landau and Vishkin's algorithm [Landau89]. But their use of a suffix tree causes significant overhead and resulted in a higher time complexity of $O(k+\log n)$.

The goal of this paper is to show that the overall time complexity of $O(k)$ can be achieved, at the expense of using more processors.

Suppose we have $m \times n$ processors, $P_{i,j}$, $(1 \leq i \leq m,\ 1 \leq j \leq n)$, where $P_{i,j}$ looks at pattern element A_i and text element T_j.

The objective is to speed up the inner while loop (statement 2 in algorithm 1).

Let *row_init* denote the initial value of the variable *row* computed by the max function just before the while loop. What the while loop does is essentially the following computation:

$$L[d,e] \leftarrow \min\{row : row = m \quad \text{OR } (row \geq row_init \text{ AND } A_{row+1} \neq T_{row+d+1})\ \} \tag{1}$$

That is, the while loop goes down $D-diagonal\ d$, starting with the cell whose row number is *row_init*, until hitting a cell with a mismatch between its corresponding A and T elements. Then $L[d,e]$ is set to the row number just before the mismatch.

In the above formula, this is expressed as a computation of finding a minimum among at most m row numbers. More specifically, in our situation, since each cell on a $D-diagonal$ has a corresponding processor, in doing the minimum finding, we have, as resources, the same number of processors as the number of row numbers. Can this task of finding minimum among at most m numbers be accomplished in $O(1)$ time?

We assume a *priority* CRCW-PRAM model (assumption b). with an indexing of the processors that is compatible with the row numbers of the processors. Specifically, we require that if $i < i'$, then $index(P_{i,j}) < index(P_{i',j})$. To determine the minimum value in formula (1) for $D - diagonal\ d$, we let each $D - diagonal\ d$ processor $P_{i,j}$ with $row\_init_d < i$ and $A_{i+1} \neq T_{i+1+d}$ write its i value to $L[d, e]$. In the *priority* model assumed, the value written has to be the minimum of those i-values.

Thus, computation in (1) can be done in $O(1)$ time trivially. One gets a simple CRCW $O(k)$ algorithm using $(n - m + k + 2) \times m$ processors.

The next algorithm implements the above idea.

Algorithm 2: (based on algorithm 1)

Initialization:

for $0 \leq d \leq n$ do $L[d, e] \leftarrow -1$
for $D - diagonal\ d, 0 \leq d \leq n$ parallel-do
    $L[d, e] \leftarrow -1$
for $D - diagonal\ d, -(k + 1) \leq d \leq -1$ parallel-do
    $L[d, |d\,| - 1] \leftarrow |d\,| - 1$
    $L[d, |d\,| - 2] \leftarrow |d\,| - 2$
for $-1 \leq e \leq k$ parallel-do
    $L[n + 1, e] \leftarrow -1$

Main loop:

for $e = 0$ to k do
    for $D - diagonal\ d, -e \leq d \leq n$ parallel-do
        $row\_init_d \leftarrow \max[(L[d, e - 1] + 1), (L[d - 1, e - 1]), (L[d + 1, e - 1] + 1)]$
        $L[d, e] \leftarrow \min(row\_init_d, m)$
    if $L[d, e] < m$ then
        for $P_{i,i+d}$, $row\_init_d \leq i \leq m$ parallel-do
            if $i \neq m$ then

```
        if $A_{i+1} \neq T_{i+1+d}$ then
            $L[d,e] \leftarrow i$                    //concurrent write
    else
        $L[d,e] \leftarrow i$                        //concurrent write
if $L[d,e] = m$ then
    print "There is an occurrence ending at $T_{d+m}$"
```

We assume that the algorithm is synchronized in such a way that the two lines labeled "concurrent write" execute at the same time.

Note that, except for the concurrent write statements, processors only access local memory and the memory of neighboring processors (assuming that the two strings have been preloaded into the processor array). This would be very important for a VLSI implementation. In addition, the $m-1$ extra processors (beyond the first) associated with a *D-diagonal* can be very simple–their only job is to enable finding the minimum index of an activated processor in $O(1)$ time. Thus, the algorithm should be well-suited to a VLSI implementation.

Similarly, one can easily show that with assumption c, i.e., the *combination* model, one can also achieve the $O(1)$ minimum-finding goal in a similar way. To do so, one needs, within the *combination* model, to assume that the combination value written will always be the minimum number among all the numbers being written.

3 $O(k)$ time in the common model with $O(n \times m \times m)$ processors

The same worst-case time complexity bound, $O(k)$, can also be achieved with assumption d, the common-CRCW model. But one has to use many more processors.

For taking minimum among at most m numbers, we can use an existing $O(1)$-time-complexity algorithm which uses $m \times m$ processors. See page 704 of [Cormen90]. They describe a parallel algorithm which computes the maximum of m numbers in $O(1)$ time using $m \times m$ processors in a common-CRCW model. For our situation, we need $m \times m$ processors for each $D - diagonal$. This increases the number of processors needed by a factor of m. Thus, one gets an $O(k)$ CRCW algorithm that uses $(n-m+k+2)\times m\times m$ processors.

Since this idea seems only have theoretical value, and since the resulting algorithm would be rather tedious, we will not give one here. But the idea should be quite clear.

4 Conclusion

This paper shows several ways in which one can get an $O(k)$ time algorithm for the *k difference approximate string matching problem*, each being associated with a specific CRCW-PRAM model assumption.

References

[Almasi89] G. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin Cummings, Redwood City, CA, 1989.

[Apostolico88] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin, Parallel construction of a suffix tree with applications, *Algorithmica*, **3**, (1988), 347–365.

[Baeza-Yates92] R. Baeza-Yates, and G. H. Gonnet, A new approach to text searching, *Comm. Assoc. Computing. Mach.*, **35** (1992), 75–82.

[Cormen90] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, (Chapter 30, Algorithms for Parallel Computers) MIT press, 1990.

[Erickson83] B. W. Erickson, and P. H. Sellers, Recognition of patterns in genetic sequences, in [Sankoff83], (1983), 55–91.

[Galil88] Z. Galil, and R. Giancarlo, Data structures and algorithms for approximate string matching, *J. Complexity*, **4** (1988), 33–72.

[Galil90] Z. Galil, and K. Park, An improved algorithm for approximate string matching, *SIAM J. Comput.*, **19**, No. 6 (1990), 989–999.

[Guibas79] L. J. Guibas, H. T. Kung, and C. D. Thompson, Direct VLSI implementation of combinatorial algorithms, *Proc. Conf. VLSI: Architecture, Design, and Fabrication,* California Institute of Technology, Pasadena, CA, 1979, 509–525.

[Landau85] G. M. Landau, and U. Vishkin, Efficient string matching in the presence of errors, *Proceedings, 26th IEEE FOCS*, 1985, 126–136.

[Landau86] G. M. Landau, and U. Vishkin, Efficient string matching with k mismatches. *Theoretical Computer Science*, **43** (1986), 239–249.

[Landau88] G. M. Landau, and U. Vishkin, Fast string matching with k differences, *J. Comput. System Sci.*, **37** (1988), 63–78.

[Landau89] G. M. Landau, and U. Vishkin, Fast parallel and serial approximate string matching, *J. Algorithms*, **10** (1989), 157–169.

[Lander91] E. S. Lander, R. Langridge, and D. M. Saccocio, Mapping and interpreting biological information, *Communications of The ACM*, **34**, No. 11 (1991), 33–39.

[Leighton92] F. Thomson Leighton, *Introduction to Parallel Algorithms and Architectures*, (Section 3.6, Simulating a Parallel Random Access Machine) Morgan Kaufmann Publishers, 1992.

[Manber89] U. Manber, *Introduction to Algorithms, a creative approach*, Addison-Wesley, 1989, pp. 155–158.

[Needleman70] S. B. Needleman, and C. D. Wunsch, A general method applicable to the search for similarities in the amino acid sequences of two proteins, *J. Mol. Bio.*, **48** (1970), 444–453.

[Sankoff83] D. Sankoff, J. B. Kruskal (Eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, 1983.

[Sellers74] P. H. Sellers, An algorithm for the distance between two finite sequences, *J. Combinator. Theor.*, A16 (1974), 253–258.

[Sellers79] P. H. Sellers, Pattern recognition in genetic sequences, *Pro. Natl Acad. Sci. USA*, **76** (1979), 3041.

[Sellers80] P. H. Sellers, The theory and computation of evolutionary distances; pattern recognition. *J. Algorithms*, **1** (1980) 359–373.

[Ukkonen83] E. Ukkonen, On approximate string matching, *in Proceedings Int. Conf. Found. Comput. Theory*, *Lecture Notes in Computer Science*, **158** (1983), 487–495.

[Ukkonen85a] E. Ukkonen, Finding approximate pattern in strings, *J. Algorithms*, **6** (1985), 132–137.

[Ukkonen85b] E. Ukkonen, Algorithms for approximate string matching, *Inform. and Control*, **64** (1985), 100–118.

[Wagner74] R. A. Wagner, and M. J. Fischer, The string-to-string correction problem, *J. ACM*, **21** (1974), 168–173.

[Wu92] S. Wu, and U. Manber, Fast text searching allowing errors, *Comm. Assoc. Computing. Mach.*, **35** (1992), 83–91.