Length of Maximal Common Subsequences

Kim S. Larsen
kslarsen@daimi.aau.dk

Computer Science Department, Aarhus University
Ny Munkegade, 8000 Aarhus C, Denmark

October 1992

Abstract

The problem of computing the length of the maximal common subsequences of two strings is quite well examined in the sequential case. There are many variations, but the standard approach is to compute the length in quadratic time using dynamic programming. A linear-time parallel algorithm can be obtained via a simple modification of this strategy by letting a linear number of processors sweep through the table created for the dynamic programming approach.

However, the contribution of this paper is to show that the problem is in NC. More specifically, we show that the length of the maximal common subsequences of two strings s and t can be computed in time $O(\log|s| \cdot \log|t|)$ in the CREW PRAM model and in time $\Theta(\min(\log|s|, \log|t|))$ in the COMMON CRCW PRAM model.

1 Introduction

A subsequence of a string s is any string, which can be created from s by deleting some of the elements. More precisely, if s is the string $s_1 s_2 \cdots s_k$ then $s_{i_1} s_{i_2} \cdots s_{i_p}$ is a subsequence of s if $\forall j \in \{1, \ldots, p\} : i_j \in \{1, \ldots, k\}$

and $\forall j \in \{1, \ldots, p-1\} : i_j < i_{j+1}$. For example, *parle* is a subsequence of *parallel*.

Consider two fixed strings $s = s_1 s_2 \cdots s_k$ and $t = t_1 t_2 \cdots t_m$. Among the strings which are subsequences of both s and t, there will be some of maximal length. Such subsequences are called *maximal common subsequences*.

For example, *parallel* and *peal* have maximal common subsequences *pal* and *pel*. However, the length of the maximal common subsequences is unique; in this case, it is three. Let *count(s,t)* denote the length of the maximal common subsequences of s and t.

The problem of computing *count(s,t)* is quite well examined in the sequential case. The solutions in the literature are variations over the basic dynamic programming approach of filling out a table with *count* values for prefixes of the argument strings. In the next section, we present one of these algorithms, which has time complexity $\Theta(|s| \cdot |t|)$. Faster algorithms can be obtained for some special cases such as: short second argument, nearly identical strings, etc. A summary of these results can be found in [1, 4]. In this paper, the focus is on the parallel time complexity.

All algorithms presented here will be given using C. The code will be almost complete. We will leave out some of the most trivial details, though. When describing parallel algorithms, a few extra primitives are needed. We assume that processes can be declared using the keyword `process`, and that each processor has an `id` variable. Processors are assumed to be numbered from one. For synchronization purposes, we use the keyword `await` followed by a boolean expression. This could have been written using a busy loop. However, we prefer to high-light the synchronization points by using a special keyword. Finally, variables not declared in the processes are assumed to be global.

2 A Sequential Solution

For completeness, we now describe one of the standard sequential solutions for computing *count(s,t)*. We assume that s is in $s[1], s[2], \ldots, s[k]$ and that t is in $t[1], t[2], \ldots, t[m]$. A table is created and the value $count(s_1 s_2 \ldots s_i, t_1 t_2 \ldots t_j)$ will be stored in $table[i][j]$.

It turns out to be convenient to represent explicitly the fact that the value of *count(s,t)* is zero if one of the strings is empty. The zero-entries will be used for that. This decision will cut down on special cases in the rest of the algorithm. This border is first filled out, and then the remaining entries are computed from top to bottom and from left to right.

This formulation of the algorithm is from [2], except that we are not concerned with the space complexity issues. The algorithm has time complexity $\Theta(|s| \cdot |t|)$.

```
char s[k+1], t[m+1];
int table[k+1][m+1];

int count()
{
   int i,j;

   for (i = 0; i <= k; i++) table[i][0] = 0;

   for (j = 0; j <= m; j++) table[0][j] = 0;

   for (j= 1; j <= m; j++)
      for (i = 1; i <= k; i++)
         table[i][j] =
            (s[i] == t[j] ?  1+table[i-1][j-1]
                          :  max(table[i][j-1],table[i-1][j]));

   return table[k][m];
}
```

3 Parallel Solutions

The dynamic programming solution presented in the previous section is the starting point for the parallel versions. In this section, we first present the most obvious generalization. This solution is included primarily in order to present the framework on a simple example. The second solution is more involved. Here, we are trying to get the asymptotic parallel time complexity down as far as possible.

3.1 A Simple Parallel Solution

An easy observation from the sequential case is that the value of $table[i][j]$, independently of whether or not $s[i]$ equals $t[j]$, is computed from table entries $table[i'][j']$, where $i' + j' < i + j$. This means that a parallel version can be designed by letting a number of processors sweep diagonally down the table.

As in the sequential dynamic programming solution, we use one row and one column to explicitly represent the fact that a count involving an empty string gives zero. The processors share the responsibility for initializing these extra entries.

The "sweep" now works as follows. We use m processors numbered from one. Processor j will be responsible for computing all the entries in the jth row. In the first step, processor 1 computes $table[1][1]$. In the second step, processor 1 computes $table[2][1]$ while processor 2 computes $table[1][2]$. In the third step, the processors 1, 2, and 3 compute the entries $table[3][1]$, $table[2][2]$, and $table[1][3]$, respectively. Proceeding like this, the whole table will be filled out in $|s| + |t| - 1$ steps. Notice that just as some processors start late, some processors finish early.

In each step, each processor performs a constant amount of work. This can easily be bounded. We assume that the global variable *step* is incremented by a designated processor using this bound. All m processors have the same code, which depend on their *id* variable:

```
char s[k+1], t[m+1];
int table[k+1][m+1], step;

process
{
   int next, i;

   table[0][id] = 0
   for (i = id−1; i <= k; i+ = m) table[i][0] = 0;

   next = id;
   while (next-id+1 <= k){
         await (next == step);
         i = next−id+1;
         table[i][j] =
         (s[i] == t[j]    ?  1+table[i−1][id−1]
                          :  max(table[i][id−1],table[i−1][id]));
   next++;
   }
}
```

The result of the computation can be found in *table*$[k][m]$. We have demonstrated that *count(s,t)* can be computed in time $\Theta(|s| + |t|)$ on a CREW PRAM (or even an EREW PRAM).

3.2 A Fast Parallel Solution

By only storing *count* values, as we did in the previous section, it does not seem possible to get below $\Theta(|s|+|t|)$ because of the dependencies in the table. In the following, we store more information. Each table entry, *table*$[i][j]$, will now be a table of size $|t| + 1$ instead of simply an integer.

The processors contain a loop, which will be executed at most $\log |s| + 1$ times. We assume that $|s|$ is a power of two (if not, then we can always extend s using a special symbol not appearing elsewhere in the strings; this cannot affect the value of *count*). As in the previous algorithm, a global variable *step* is used to count the number of iterations of these loops. At a certain point, given the value of *step*, we are only interested in the following substrings of s:

$$s_1 \cdots s_{2^{step}},\ s_{2^{step}+1} \cdots s_{2 \cdot 2^{step}},\ s_{2 \cdot 2^{step}+1} \cdots s_{3 \cdot 2^{step}} \cdots,\ |s|.$$

In that round, only the entries *table*$[i][j][v]$, where $i \in \{2^{step}, 2 \cdot 2^{step}, 3 \cdot 2^{step}, \ldots, |s|\}$ are maintained.

The basic idea is that $p = table[i][i][v]$ should be the maximal index (giving rise to the shortest substring of t) such that

$$count(s_{i-2^{step}+1} \cdots s_i, t_{p+1} \cdots t_j) = v$$

Notice that the substring of t starts with t_{p+1} rather than with t_p. This results in a more elegant algorithm, as we need not worry about t_p being matched with two elements from s, when combining the values for two adjacent substrings of s into a value for the concatenation.

In the algorithm, we assign a processor to each $table[i][j][v]$ entry. Each of these processors controls their own collection of processors to compute the maxima they need. Exactly how many processors are needed for this depends on how fast we want the maxima computed and in which model. We return to the complexity issues later.

We assume that in each step, all processors have time to read all their entries before any processor writes. Apart from computing maxima, the computation strategy of which we have not specified, we are within the CREW model of computation, as only processor (i, j, v) writes in $table[i][j][v]$.

```
/* Algorithm Fast Counting */

char s[k+1], t[m+1];
int table[k+1][m+1]t[m+1];
int step;

process (i,j,v) /* 1<=i<=k, 0<=j<=m, 0<=v<=m */
{
   int h, next;

   if (v == 0)
         table[i][j][v] = j
   else if (v == 1)
         table [i][j][v] =
               max ({−1} ∪ {u ∈ {0, . . . , j − 1} | s[i] = t[u + 1]});
   else
         table[i][j][v] = −1
```

```
    h $= 1$;
    next $= 1$;
    while  (i% $(2*h) == 0)\{$
            await (next == step);

            table[i][j] =
                  $\max\ (\{-1\} \cup \{q \mid u \in \{0, \ldots, v\} \wedge$
                                 $p = table[i][j][u] \wedge p \neq -1 \wedge$
                                 $q = table[i-h][p][v-u] \wedge q \neq -1\});$

            h $= 2*$h
            next++;
    }
}
```

For the purpose of giving a correctness proof, we now formulate and prove the invariant, which was outlined intuitively before presenting the algorithm.

Lemma 1 The following loop invariant holds for the values of *step* encountered in the algorithm:
for $i \in \{c \cdot 2^{step} \mid c \in I\!N \backslash \{0\} \wedge c \cdot 2^{step} \leq |s|\}$, and $j, v \in \{0, \ldots, |t|\}$, we have that $table[i][j][v]$ equals

$$\max(\{-1\} \cup \{r \mid 0 \leq r \leq j \wedge count(s_{i-2^{step}+1} \cdots s_i, t_{r+1} \cdots t_j) = v\}).$$

Proof First, we prove that the invariant holds after the initialization. As $step = 0$, we consider all i's in $\{1, \ldots, |s|\}$. In the following, assume that i and j are fixed. Notice that $s_{i-2^{step}+1} \cdots s_i$ is simply s_i. For each v, we determine the maximal r such that $r \leq j$ and $count(s_i, t_{r+1} \cdots t_j) = v$.

Assume that $v = 0$. As $count(s_i, \varepsilon) = 0$, we can choose $r = j$ (ε denotes the empty string).

Assume that $v = 1$. In order for $count(s_i, t_{r+1} \cdots t_j)$ to equal one, s_i must appear in $t_1 \cdots t_j$. If it does not, then the maximum is -1 and $table[i][j][1]$ is also assigned -1. Now, assume that s_j is present in $t_1 \cdots t_j$. Let t_u be the right-most occurrence of s_i in $t_1 \cdots t_j$. Then $count(s_i, t_u \cdots t_j) = 1$ and u is the maximal integer for which this is the case. So, we can choose $r = u - 1$.

Assume that $v > 1$. As s_i has length one, $count(s_i, t_{r+1} \cdots t_j)$ can be at most one no matter what the value of r is. So, in this case, $table[i][j][u]$ should be assigned -1.

Now, we turn to the induction step. Let $step' = step + 1$. Assume that the maximum is -1, i.e., no r exists such that

$$count(s_{i-2^{step'}+1} \cdots s_i, t_{r+1} \cdots t_j) = v.$$

This means that even $count(s_{i-2^{step'}+1} \cdots s_i, t_1 \cdots t_j) < v$. Now, assume to the contrary that a u exists such that $p \neq -1$ and $q \neq -1$. By the invariant, this means that $count(s_{i-2^{step'}+1} \cdots s_i, t_{p+1} \cdots t_j) = u$ and that $count(s_{i-2^{step'}+1} \cdots s_{i-2^{step}}, t_{q+1} \cdots t_p) = v - u$. But the two substrings of s are disjoint as are the two substrings of t, so by definition of count, we must have that $count(s_{i-2^{step'}+1} \cdots s_i, t_{q+1} \cdots t_j) = v$, which is a contradiction.

Now, assume that the maximum is $r \geq 0$. By dividing a maximal common subsequence of $s_{i-2^{step'}+1} \cdots s_i$ and $t_{r+1} \cdots t_j$ up into the part which is in $s_{i-2^{step'}+1} \cdots s_{i-2^{step}}$ and the part in $s_{i-2^{step'}+1} \cdots s_i$ it is clear that a u and an l must exist such that

$$count(s_{i-2^{step}+1} \cdots s_i, t_{l+1} \cdots t_j) = u$$

and

$$count(s_{i-2^{step'}+1} \cdots s_{i-2^{step}}, t_{r+1} \cdots t_l) = v - u$$

By the invariant, p (in the algorithm) is the maximal integer such that $count(s_{i-2^{step}+1} \cdots s_i, t_{p+1} \cdots t_j) = u$, so obviously $l \leq p$. But then

$$count(s_{i-2^{step'}+1} \cdots s_{i-2^{step}}, t_{r+1} \cdots t_p)$$

is at least $v - u$. By the invariant, q (in the algorithm) is now the maximal integer such that $count(s_{i-2^{step'}+1} \cdots s_{i-2^{step}},\ t_{q+1} \cdots t_p) = v - u$. So, given that we insist on matching exactly u elements from $s_{i-2^{step}+1} \cdots s_i$ with elements from t, q must also be the maximal integer such that $count(s_{i-2^{step'}+1} \cdots s_i,$ $t_{q+1} \cdots t_j) = v$. Because in order to use a larger q, we would have to use a larger p, and that would make

$$count(s_{i-2^{step}+1} \cdots s_i, t_{p+1} \cdots t_j)$$

strictly smaller than u, as p was chosen to be maximal.

As all u's are considered in the algorithm, and as $s_{i-2^{step}+1} \cdots s_i$ must match u elements from t for some $u (0 \leq u \leq v)$, we are bound to find the right u and, thus, the correct maximum value. □

Corollary 2 Algorithm *Fast Counting* correctly computes *count(s,t)* when the two strings s and t are used in the algorithm and the final result is computed from the table by

$$count(s,t) = \max_{0 \leq v \leq m} \{v \mid table[k][m][v] \neq -1\}.$$

Proof The last processors terminate when $step = \log k$, and at that point,

$$table[k][m][v] = \max(\{-1\} \cup \{r \mid 0 \leq r \leq m \wedge count(s, t_{r+1} \cdots t_m) = v\}).$$

Clearly, $table[k][m][v] = -1$ if and only if $count(s,t) < v$. □

Finally, we consider the complexity issues. Recall that in the CREW PRAM model, there are a polynomial number of processors and a shared memory. Any number of processors can read from the same memory cell at the same time, but at most one processor can be writing to any memory cell at a given time. The COMMON model is a CRCW PRAM model. The CRCW PRAM is like the CREW PRAM except that concurrent writes are allowed. In the COMMON model, if more than one processor writes to the same memory cell at the same time, then they must all write the same value. The COMMON model has also been called a WRAM in the literature.

The algorithm has been designed such that maxima computations and the remaining computation can be analyzed separately. In the following, we state the known maxima results of interest here.

Lemma 3 The problem of calculating the maximum of n elements is in NC, and the maximum can be computed in time

- $O(\log n)$ in the CREW model.
- $\Theta(1)$ in the COMMON CRCW model.

Proof In the CREW model, use a binary tree of height $\lceil \log n \rceil$.
In the COMMON model, $\binom{n}{2}$ processors are used to perform every possible comparison in one step. The details can be found in [3].

Only comparisons, assignments, additions, and subtractions are used in the algorithms, so the problem is clearly in NC. $\Box$

Theorem 4 The complexity of calculating the length of the maximal common subsequences is

- $O(\log |t| \cdot \log |s|)$ in the CREW model.
- $\Theta(\min(\log |t|, \log |s|))$ in the COMMON model.

Proof Rename, if necessary, so that $|s| \leq |t|$. The algorithm runs in $\log |s| + 1$ steps and, except for the calculation of maxima, only a constant amount of work is done by each processor in each step. Notice that the sets for which we find the maxima have size at most $|t| + 2$. Furthermore, the predicates used in their definition can be computed in constant time. The result now follows from lemma 3. $\Box$

Corollary 5 The problem of computing the length of the maximal common subsequences is in NC.

Proof All computations, except the maxima computations, are independent of the input length. As stated in lemma 3, computing the maximum is in NC, so our problem is as well. $\Box$

Notice that since computing the maximum is in NC^1, the problem of computing the length of the maximal common subsequences is in NC^2. Using unbounded fan-in, the maximum can be computed in constant time, so the problem of computing the length of the maximal common subsequences is also in AC^1.

Acknowledgement The author would like to thank Joan Boyar for valuable discussions on the models of parallel computation and for helpful comments on an earlier draft of the paper.

References

[1] Alfred V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science,* pages 255-300. Elsevier Science Publishers, 1990.

[2] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Comm. ACM,* 18(6):341-343, 1975.

[3] Yossi Shiloach and Uzi Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *J. Algorithms,* 2:88-102, 1981.

[4] Graham A. Stephen. String search. Technical Report TR-92-gas-01, School of Electronic Engineering Science, University College of North Wales, 1992.