Derivation of a Parallel String Matching Algorithm

Jayadev Misra
The University of Texas at Austin
Austin, Texas 78712, USA
email: misra@cs.utexas.edu

July 28, 2001

Abstract

We derive an efficient parallel algorithm to find all occurrences of a *pattern* string in a *subject* string in $O(\log n)$ time, where n is the length of the subject string. The number of processors employed is of the order of the product of the two string lengths. The theory of powerlists [2, 3] is central to the development of the algorithm and its algebraic manipulations.

1 Introduction

We describe a parallel algorithm that finds all occurrences of a *pattern* string in a *subject* string in $O(\log n)$ time, where n is the length of the subject string. The number of processors employed is of the order of the product of the two string lengths. The emphasis in this note is on the derivation of the algorithm. First, a simple version is derived whose correctness is obvious and next, a more sophisticated version is derived through algebraic manipulations of the original program. The theory of powerlists [2, 3] is central to the development of the algorithm and its algebraic manipulations.

The proposed algorithm is more general than "exact" string matching. A "match" can be any binary relation over the symbols of the given alphabet. Therefore, wild character matches as well as characters that never match may be used in either string.

2 Problem Description

We are given two strings, a pattern called f and a subject called g, over some alphabet. Henceforth, the length of a string h is denoted by $|h|$. We assume that $|f|$ and $|g|$ are both powers of 2; we discuss in section 7 how this restriction can be removed. The symbols in a string are indexed starting at 0 and we write h_i to refer to the symbol with index i in string h.

It is required to output a string of boolean values b of the same length as the subject, denoting for each position if the pattern matches the subject starting at that position. That is,

$$
\begin{array}{ll}
b_i = (\forall j : 0 \leq j < |f| : f_j = g_{i+j}), & 0 \leq i \leq |g| - |f| \\
b_i = \mathit{False}, & |g| - |f| < i < |g|
\end{array}
$$

As we have remarked earlier, $f_j = g_{i+j}$ may be replaced by $f_j \ \triangleright \ g_{i+j}$ for any binary relation $\triangleright$.

3 Powerlist

The *powerlist* data structure was introduced in [3] to facilitate descriptions of parallel algorithms. The smallest powerlist —corresponding to the empty list for the linear case— is a list of one element. There are two different ways in which two powerlists are joined to create a longer powerlist. If p and q are powerlists of the same length then

- $p \mid q$ is the powerlist formed by concatenating p and q, and
- $p \bowtie q$ is the powerlist formed by successively taking alternate items from p and q, starting with p.

Thus, the length of $p \mid q$ or $p \bowtie q$ is double the length of p (and q). Hence, the length of a powerlist is 2^n, for some n, $n \geq 0$. Powerlists can be nested, but we will not use that feature in this note.

In the following examples the sequence of elements of a powerlist are enclosed within angular brackets.

$$
\begin{array}{l}
\langle 0 \rangle \mid \langle 1 \rangle = \langle 0\ 1 \rangle,\ \langle 0 \rangle \bowtie \langle 1 \rangle = \langle 0\ 1 \rangle, \\
\langle 0\ 1 \rangle \mid \langle 2\ 3 \rangle = \langle 0\ 1\ 2\ 3 \rangle,\ \ \langle 0\ 1 \rangle \bowtie \langle 2\ 3 \rangle = \langle 0\ 2\ 1\ 3 \rangle
\end{array}
$$

The operator $\mid$ is called *tie* and $\bowtie$ is *zip*. Note that $(p \bowtie q)_{2j} = p_j$ and $(p \bowtie q)_{2j+1} = q_j$, for all j, $0 \leq j < |p|$.

Convention: We write function application without parentheses where no confusion is possible. For functions d and e, we write "$d\ x$ instead of "$d(x)$" and "$e\ x\ y$" instead of "$e(x,y)$". The constructors $\mid$ and $\bowtie$ have the same binding power and their binding power is lower than that of function application. $\Box$

Functions over linear lists are typically defined by case analysis —a function is defined over the empty list and, recursively, over non-empty lists. Functions over powerlists are defined analogously. For instance, the following function reverses the order of the elements of the argument powerlist.

$$
\begin{array}{l}
rev\langle x \rangle = \langle x \rangle \\
rev(p \mid q) = (rev\ q) \mid (rev\ p)
\end{array}
$$

The case analysis, as for linear lists, is based on the length of the argument powerlist. We adopt the pattern matching scheme of modern functional programming languages, such as Haskell [1], to *deconstruct* the argument list into its components, p and q, in the recursive case. Deconstruction, in general, uses the operators $\mid$ and $\bowtie$. In the definition of rev, we have used $\mid$ for deconstruction; we could have used $\bowtie$ instead and defined rev in the recursive case by

$$rev(p \bowtie q) = (rev\ q) \bowtie (rev\ p)$$

It can be shown that the two proposed definitions of rev are equivalent and the following equation holds for any powerlist p.

$$rev(rev\ p) = p$$

Note: In this paper, we use only the zip operator over powerlists. Each string is treated as a powerlist of symbols.

4 A Simple Pattern Matching Algorithm

We define function sm that matches pattern f over subject g. If f and g are both singleton lists, say $\langle x \rangle$ and $\langle y \rangle$ respectively, then the output is $\langle True \rangle$ if $x = y$ and $\langle False \rangle$ otherwise. That is,

$$sm\ \langle x \rangle\ \langle y \rangle = \langle x = y \rangle$$

For f a singleton and g a non-singleton, the same strategy can be applied recursively.

$$sm\ \langle x \rangle\ (r \bowtie s) = (sm\ \langle x \rangle\ r)\ \bowtie\ (sm\ \langle x \rangle\ s)$$

For f a non-singleton list and g a singleton, the output is $\langle False \rangle$.

$$sm\ (p \bowtie q)\ \langle y \rangle = \langle False \rangle$$

Next, consider the general case where f is of the form $(p \bowtie q)$ and g is $(r \bowtie s)$. We say that f matches g at index j if $(\forall k : 0 \le k < |f| : f_k = g_{j+k})$.

- Assertion 1: $p \bowtie q$ matches $r \bowtie s$ at some even index $2k$ iff p matches r at index k and q matches s at index k.
- Assertion 2: $p \bowtie q$ matches $r \bowtie s$ at some odd index $2k + 1$ iff p matches s at index k and q matches r at index $k + 1$.

We sketch a proof of one part of assertion 2.

$$
\begin{array}{ll}
& p \bowtie q \text{ matches } r \bowtie s \text{ at } 2k+1 \\
\equiv & \{\text{definition of "matches"}\} \\
& (\forall k : 0 \le j < |p \bowtie q| : (p \bowtie q)_j = (r \bowtie s)_{j+2k+1}) \\
\Rightarrow & \{\text{consider only the odd indices } 2j+1\} \\
& (\forall j : 0 \le 2j+1 < |p \bowtie q| : (p \bowtie q)_{2j+1} = (r \bowtie s)_{2j+1+2k+1}) \\
\Rightarrow & \{(p \bowtie q)_{2j+1} = q_j, (r \bowtie s)_{2j+1+2k+1} = r_{j+k+1}\} \\
& (\forall j : 0 \le j < |q| : q_j = r_{j+k+1}) \\
\Rightarrow & \{\text{definition of "matches"}\} \\
& q \text{ matches } r \text{ at } k+1
\end{array}
$$

These two assertions permit computation of $sm\ (p \bowtie q)\ (r \bowtie s)$ from $(sm\ p\ r)$, $(sm\ q\ s)$, $(sm\ p\ s)$, and $(sm\ q\ r)$. First abbreviate $(sm\ x\ y)$ by $smxy$ for all x and y. Assertion 1 says that the sublist of $sm\ (p \bowtie q)\ (r \bowtie s)$ with even indices is $(smpr\ \wedge\ smqs)$, where $\wedge$ is applied pointwise on $smpr$ and $smqs$.

Unlike assertion 1, the corresponding indices in assertion 2 (k and $k+1$) are not identical. Therefore, the sublist of $sm\ (p \bowtie q)\ (r \bowtie s)$ with odd indices is not merely $(smqr\ \wedge\ smps)$; the indices in $smqr$ have to be decremented, i.e., $smqr$ has to undergo a left shift. Let $sm'qr$ denote the left shift of $smqr$ where *False* is added to the right end of the list when it is shifted left. The the sublist of $sm\ (p \bowtie q)\ (r \bowtie s)$ with odd indices is $(sm'qr\ \wedge\ smps)$. The entire list $sm\ (p \bowtie q)\ (r \bowtie s)$ is obtained by zipping its sublists with even and odd indices. We give the complete algorithm below including a definition of left shift.

$$
\begin{array}{ll}
sm\ \langle x\rangle\ \langle y\rangle = \langle x = y\rangle & \\
sm\ \langle x\rangle\ (r \bowtie s) = (sm\ \langle x\rangle\ r)\ \bowtie\ (sm\ \langle x\rangle\ s) & \\
sm\ (p \bowtie q)\ \langle y\rangle = \langle \mathit{False}\rangle & \\
sm\ (p \bowtie q)\ (r \bowtie s) = & (smpr\ \wedge\ smqs)\ \bowtie\ (sm'qr\ \wedge\ smps) \\
& \text{where} \\
& \quad smpr = sm\ p\ r \\
& \quad smqs = sm\ q\ s \\
& \quad sm'qr = ls(sm\ q\ r) \\
& \quad smps = sm\ p\ s
\end{array}
$$

The definition of left shift, ls, is

$$
\begin{array}{l}
ls\ \langle x\rangle = \langle \mathit{False}\rangle \\
ls\ (u \bowtie v) = v\ \bowtie (ls\ u)
\end{array}
$$

5 An Improved Pattern Matching Algorithm

In a parallel implementation $\bowtie$ and $\wedge$ over lists can be taken to be constant time operations if each list element is stored at an appropriate processor. Then we have described a $O(\log|f| \times \log|g|)$ time algorithm for matching pattern f over subject g. To see this, first note that left shift applied to a list of length k takes $O(\log k)$ parallel time. The algorithm for sm first takes $O(\log|f|)$ steps each requiring $O(\log|g|)$ time, since a left shift is applied at every step; then the pattern is reduced to a singleton and thereafter $O(\log|g| - \log|f|)$ steps each needing constant time are required.

In this section, we show an optimization that reduces the parallel running time to $O(\log|g|)$. The optimization involves eliminating the left shift function, and we do this through purely algebraic manipulations.

Define function sm' where $sm'\ u\ v$ is $ls(sm\ u\ v)$; this is written as $sm'uv$ according to our notational convention. We derive the definition of sm' from those of sm and ls.

$$\begin{array}{cl}
 & sm'\ \langle x\rangle\ \langle y\rangle \\
= & \{\text{definition of } sm'\} \\
 & ls(sm\ \langle x\rangle\ \langle y\rangle) \\
= & \{\text{definition of } sm\ \langle x\rangle\ \langle y\rangle\} \\
 & ls\langle x = y\rangle \\
= & \{\text{definition of } ls \text{ on a singleton list}\} \\
 & \langle \mathit{False}\rangle
\end{array}$$

$$\begin{array}{cl}
 & sm'\ \langle x\rangle\ (r \bowtie s) \\
= & \{\text{definition of } sm'\} \\
 & ls(sm\ \langle x\rangle\ (r \bowtie s)) \\
= & \{\text{definition of } sm\ \langle x\rangle\ (r \bowtie s)\} \\
 & ls((sm\ \langle x\rangle\ r)\ \bowtie\ (sm\ \langle x\rangle\ s) \\
= & \{\text{definition of } ls\ (u \bowtie v)\ \} \\
 & (sm\ \langle x\rangle\ s)\ \bowtie\ ls(sm\ \langle x\rangle\ r) \\
= & \{\text{definition of } sm'\} \\
 & (sm\ \langle x\rangle\ s)\ \bowtie\ (sm'\ \langle x\rangle\ r)
\end{array}$$

A similar derivation shows that

$$sm'\ (p \bowtie q)\ \langle y\rangle = \langle \mathit{False}\rangle$$

Finally,

$$\begin{array}{cl}
 & sm'\ (p \bowtie q)\ (r \bowtie s) \\
= & \{\text{definition of } sm'\} \\
 & ls(sm\ (p \bowtie q)\ (r \bowtie s)) \\
= & \{\text{definition of } sm\ (p \bowtie q)\ (r \bowtie s)\} \\
 & ls((smpr\ \wedge\ smqs)\ \bowtie\ (sm'qr\ \wedge\ smps)) \\
= & \{\text{definition of } ls\} \\
 & (sm'qr\ \wedge\ smps)\ \bowtie\ ls((smpr\ \wedge\ smqs)) \\
= & \{ls \text{ distributes over } \wedge \text{ in the second term}\} \\
 & (sm'qr\ \wedge\ smps)\ \bowtie\ (ls(smpr)\ \wedge\ ls(smqs)) \\
= & \{ls(smpr) = sm'pr \text{ and } ls(smqs) = sm'qs\} \\
 & (sm'qr\ \wedge\ smps)\ \bowtie\ (sm'pr\ \wedge\ sm'qs)
\end{array}$$

The left shift operation has now been eliminated. Both sm and sm' now require $O(\log n)$ parallel time, where n is the length of the subject string.

6 Putting the Pieces Together

We give the definitions of sm and sm' together for easy reference.

$$
\begin{aligned}
&sm\ \langle x\rangle\ \langle y\rangle = \langle x = y\rangle\\
&sm\ \langle x\rangle\ (r \bowtie s) = (sm\ \langle x\rangle\ r)\ \bowtie\ (sm\ \langle x\rangle\ s)\\
&sm\ (p \bowtie q)\ \langle y\rangle = \langle \mathit{False}\rangle\\
&sm\ (p \bowtie q)\ (r \bowtie s) = \ (smpr\ \wedge\ smqs)\ \bowtie\ (sm'qr\ \wedge\ smps)\\
&\qquad \text{where}\\
&\qquad\qquad smpr = sm\ p\ r\\
&\qquad\qquad smqs = sm\ q\ s\\
&\qquad\qquad sm'qr = sm'\ q\ r\\
&\qquad\qquad smps = sm\ p\ s
\end{aligned}
$$

$$
\begin{aligned}
&sm'\ \langle x\rangle\ \langle y\rangle = \langle \mathit{False}\rangle\\
&sm'\ \langle x\rangle\ (r \bowtie s) = (sm\ \langle x\rangle\ s)\ \bowtie\ (sm'\ \langle x\rangle\ r)\\
&sm'\ (p \bowtie q)\ \langle y\rangle = \langle \mathit{False}\rangle\\
&sm'\ (p \bowtie q)\ (r \bowtie s) = \ (sm'qr\ \wedge\ smps)\ \bowtie\ (sm'pr\ \wedge\ sm'qs)\\
&\qquad \text{where}\\
&\qquad\qquad sm'qr = sm'\ q\ r\\
&\qquad\qquad smps = sm\ p\ s\\
&\qquad\qquad sm'pr = sm'\ p\ r\\
&\qquad\qquad sm'qs = sm'\ q\ s
\end{aligned}
$$

7 Concluding Remarks

The main thrust of this paper is a demonstration that the algebraic properties of powerlists make it possible to derive an algorithm systematically; the final version is efficient and it would have been difficult to arrive there intuitively. Also, observe that no specific property of the equality operator over the symbols of the alphabet has been exploited (such as that it is an equivalence relation). Therefore, the proposed algorithm is applicable where the relevant operator is any binary relation over the alphabet. Thus, if the alphabet contains symbols α and β where α "matches" every symbol in the alphabet except β and β matches no symbol then the pattern can be extended by αs and the subject by βs without affecting the relevant part of the result. This is particularly useful when the length of the pattern or the subject string is not a power of 2; then such extensions can be applied to bring their lengths to the next power of 2.

Acknowledgment: I appreciate the comments of Edsger W. Dijkstra.

References

[1] Haskell 98: A non-strict, purely functional language. Available at `http://haskell.org/onlinereport`, 1999.

[2] Jacob Kornerup. *Data Structures for Parallel Recursion.* PhD thesis, University of Texas at Austin, 1997. Available for download at `http://www.cs.utexas.edu/users/kornerup/dis.ps.Z`.

[3] Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Transactions on Programming Languages and Systems*, 16(6):1737–1767, November 1994.