A Generalisation of Indexing for Parallel Document Search

D.B. Skillicorn
skill@qucis.queensu.ca

March 1995

External Technical Report
ISSN-0836-0227-
95-383

Department of Computing and Information Science

Queen's University

Kingston, Ontario K7L 3N6

Document prepared March 21, 1995

A Generalisation of Indexing for Parallel Document Search

Abstract

Parallelism is useful in the storage and access of structured documents. Fast parallel algorithms for search in structured text are already known, but they will not supplant the use of indexes to speed up searching until massively parallel architectures become routinely available. However, parallel algorithms suggest new kinds of indexes that provide powerful search capability and performance even on modestly-parallel computers.

We present a generalisation of indexes based on regular languages, called indexing languages, that are chosen to be homomorphic images of languages generated by typical search patterns. Precomputing properties of text strings relative to indexing languages makes it fast to exclude large parts of the text from consideration before executing a direct search.

1 Background

Search times in document archives are improved by building indexes giving the location of (usually) words as offsets in the structure. This kind of index has three major drawbacks: it is expensive to compute, often taking days for moderately-sized archives; it does not permit searches that cannot be expressed as simple conjunctions and disjunctions on words; and it requires large ancillary storage, typically twice as large as the documents themselves.

A related technique is the use of signature files [1] which "condense" the set of words, or perhaps only index terms, in a document down to a single key. A similar operation is then applied to the search terms, and a simple test determines those documents that cannot contain the search terms (although it only indicates documents that *may* contain them). This reduces the storage load, but still limits searches to boolean expressions on words.

Document archives are large enough to benefit from parallel computation techniques. Techniques for parallelizing both index and signature file management have been suggested, but they have shown limited promise. However, starting from the premise that parallelism is integral to document archives, and that documents are typically stored in a distributed way and accessed by multiple processors provides a new perspective on how to index them. These insights also provide alternative approach to indexing sequentially-accessed documents.

Parallel document processing systems are capable of evaluating searches described by arbitrary regular expressions (and hence more powerful than the query capability of many existing search systems) in time logarithmic in the total size of the documents, provided that enough processing power is available [6]. This suggests that, in the long run, indexing will decrease in importance since even complex queries can be directly evaluated in reasonable amounts of time. However, it is, for the present, unreasonable to expect that parallel systems will have more than a small number of processors (say up to a hundred), while document archives hold thousands of documents and have a total size of many terabytes. Thus although indexing will continue to play a role in the medium term, new and more powerful kinds of indexes may be useful.

In this paper we show how search problems can be cast as regular language recognition problems. This suggests the use of a set of indexing languages and associated homomorphisms that capture commonality between search patterns. Precomputing information about membership in these indexing languages for segments of the text string, already partitioned and distributed across a set of processors, makes it possible to eliminate segments from further consideration quickly and cheaply. The resources of the parallel computer are then better deployed to search segments in which instances of the search pattern might appear. The techniques are described for text treated as linear strings, but the approach generalises

trivially to text structured in a tree-like way as in SGML.

2 A Different Perspective on Search

The basic search problem is this: given a string w (the search string) and a string s (the document), determine if w is present in s. Most indexes are set up to solve this problem.

However, we begin with a slight, but powerful, extension: given a regular expression r (the search pattern), and a string s (the document), determine if an instance of the pattern is present in the string. This kind of search is commonly available in text editors, but cannot be easily assisted by indexes unless the pattern is very simple. It is, however, a useful and powerful search capability. For example, it allows patterns such as occurrences of the word X inside a chapter heading to be searched for, using a pattern

$$< chapter > \Sigma^* X \Sigma^* < /chapter >$$

where Σ^* means any terminal string of length 0 or greater.

We express such searches as language recognition problems in the following way: the problem above is equivalent to asking if the string s is a member of the language L described by $\Sigma^* r \Sigma^*$, that is the language that contains all strings described by r, possibly surrounded by arbitrary strings. We call such a language the language generated by the regular expression. Thus the search problem becomes a regular language recognition problem.

There is a highly-parallel algorithm for regular language recognition due to Fischer [2] (and described for the Connection Machine in [4]). We review it briefly. Given a regular language L, there is a (deterministic) finite state automaton that accepts exactly the strings of the language. For example, if the regular expression is the string "rat" then the finite state automaton that accepts strings containing the word "rat" is shown in Figure 1.

Each terminal in the language is associated with some set of transitions in the automaton. The first step of the algorithm is to construct a set of such transitions for each terminal, that is to construct the sets

$$Transitions(x) = \{(q_i, q_j) \mid \exists \text{ a transition labelled } x \text{ from } q_i \text{ to } q_j\}$$

These sets are of bounded size independent of the size of the target string (in fact, they are bounded by the square of the number of states in the automaton, which is a function of the length of the regular expression).

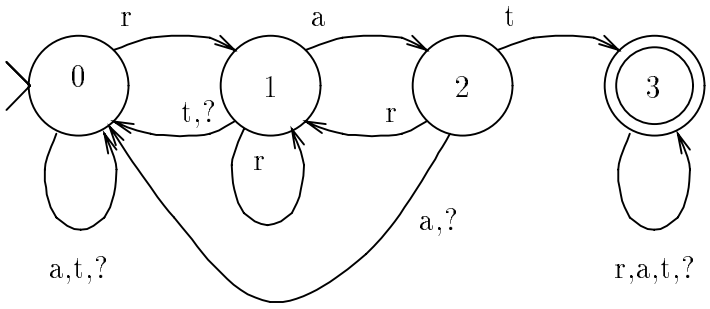

Figure 1: Finite State Automaton accepting Strings containing "rat" – ? denotes any other symbol in the alphabet

Now we define a binary operation, $\circledast$, that composes two transition sets to give a single one like this:

$$\begin{aligned} &\mathit{Transitions}(x) \circledast \mathit{Transitions}(y) = \\ &\quad \{(q_i, q_k) \mid (q_i, q_j) \in \mathit{Transitions}(x) \text{ and } (q_j, q_k) \in \mathit{Transitions}(y)\} \end{aligned}$$

It is easy to see that $\circledast$ is associative and has an identity, the transition table mapping each state to itself. A composition of transition sets, say those associated with terminals x and y, describes all possible transitions in the automaton on input symbols xy. Thus long compositions capture possible paths in the automaton driven by the terminal string they represent.

The recognition problem now becomes reduction of transition sets using the operator $\circledast$. Suppose that there is a processor associated with each terminal of the target string. In a first step, each such processor computes or looks up the transition set for the terminal it holds. In the second step, adjacent transition sets are repeatedly composed. The transition sets of each odd-numbered processor are first composed with those of their right hand (even-numbered) processor to give new transition sets. Only half of the original processors now hold a transition set; the remainder play no further part in the algorithm. The same process of set composition is repeated between processors still holding transition sets. After the second step, only one quarter of the processors hold transition sets. This continues until only one processor holds a transition set that reflects the state-to-state transition effects of the whole terminal string. If this set contains a pair whose first element is the initial state and whose second element is the final state, then the target string is in the language. Note that the number of active processors is reduced by half on each step, so the reduction terminates after $\lceil \log n \rceil$ steps, where n is the length of the target string. (For the remainder of the paper we assume that all sizes are powers of two to avoid the need for ceiling functions.)

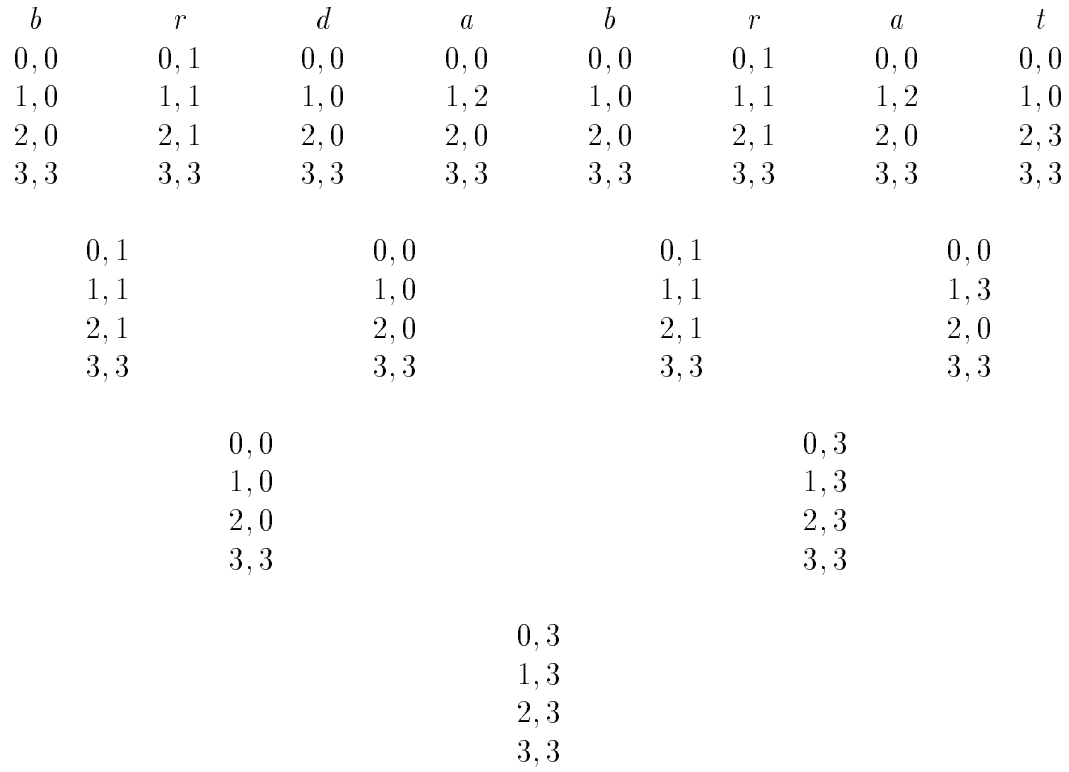

Figure 2: Progress of the Recognition Algorithm

The transition sets generated by the algorithm on a particular input string are shown in Figure 2, based on the finite state automaton in Figure 1.

The parallel time complexity of this algorithm is $\log n$ for a target string of length n, using n processors. Since n will typically be of the order of thousands or millions, parallel computers with n processors are not realistic at the moment. However, the algorithm is easy to adapt to fewer, say p processors. The target string is divided into p segments and each segment is allocated to a processor. Each processor then generates the transition sets for the terminals it contains and composes them sequentially. When each processor has produced a single transition set, describing the total effect of the substring it contains, the parallel algorithm proceeds as before on these sets, using p processors. The sequential composition of transitions sets takes time proportional to the number of sets, which is n/p, while the

parallel part of the algorithm takes time $\log p$. The total time complexity is

$$t_p = \frac{n}{p} + \log p \tag{1}$$

For small values of p this gives approximately linear speed-up, while for large p it approaches time logarithmic in the target string size. (Note that we are assuming that transition set composition takes constant time as a function of the length of the target string. The time actually depends on the number of states in the automaton, which in turn depends on the length of the search pattern. It seems reasonable to assume that this is small, but the assumption is certainly assailable.)

An important property of this algorithm is that it has no left-to-right bias built into it. After some number of time steps, a processor holding a transition set knows the effect of the substring it represents on any state in which a substring to its left might leave the automaton. This ability to derive information about a part of the target string without previous analysis of its left context is an important part of our optimisations.

Suppose that the target string s has been divided in some way across a set of processors. Call the substring of s in processor i s_i, and suppose that there are p processors. Then, after computing the transition set in each processor, we can determine locally which of the following four cases holds for s_i:

1. $s_i \in L$: This happens when an occurrence of the pattern occurs completely within substring s_i. This is detected by the presence of an entry of the form (q_0, q_f) (where q_f is the final state of the automaton) in the transition table corresponding to s_i.

2. $s_i \notin L$: This happens when no part of the pattern occurs within string s_i. This is detected because the transition set for s_i is identical to that of "other symbols" those that do not appear in the pattern, consisting of all entries of the form (q_i, q_0) except for one entry (q_f, q_f).

3. $s_i x \in L$: This happens when the transition set for s_i contains an entry of the form $(q_0, q_i)(i \neq 0)$ so that there exists some string x such that the concatenation is in the language.

4. $xs_i \in L$: This happens when the transition set for s_i contains an entry of the form $(q_i, q_f)(i \neq f)$ so that there exists a string x (whose transition set must contain an entry of the form (q_o, q_i)) such that the concatenation is in the language.

This analysis is useful in the following way: most search patterns are small (since they are human-generated) and their span in the document is also small – that is instances of the

pattern are likely to be no more than a few lines, rather than large sections of the document. Thus if the number of segments is large, almost all will be of type 2 above and need no further processing in the parallel phase of the algorithm.

The effect on the parallel time complexity given in Equation 1 above is to make it

$$t_p = \frac{n}{p} + y$$

where y depends on the particular regular expression, but is expected to be much smaller than $\log p$. This optimisation is of limited interest since today's architectures have small values of p and hence $\log p$ is already negligible. However, it sets the stage for a technique in which transition sets for the segments held in each processor are precomputed (independently of the search expression) thus reducing the first term in the time complexity of Equation 1.

3 Using Homomorphisms

Recall that a string homomorphism is a map that respects string concatenation. Thus a string homomorphism is defined by giving a function $h' : \Sigma \to \Delta^*$ and taking its closure, h, on strings.

For any string s and language L, if $s \in L$ then $h(s) \in h(L)$. The essential idea of indexing languages is to choose a set of L_j's that "collapse" some of the structure of languages generated by typical or common regular expressions, and then precompute the transition sets for segments of the text string relative to each of the L_j's. Associated with each of the L_j's is a homomorphism, h_j, whose codomain is L_j. If one of the homomorphisms maps the language generated by the search pattern to one of the indexing languages L_j, we can reason contrapositively as follows: if $h_j(s_i)$ is not a member of $L_j (= h_j(L))$ then s_i cannot be a member of L and the search pattern is not present in the segment s_i. For practical searches, this enables a large proportion of the text string to be ruled out as a possible location of the pattern early in the search, and before expending the n/p time that is required without the optimisation. Thus we choose languages $L_1, L_2, \ldots$ that represent useful simplifications of structures in the text string that are expected to play a role in searches. The transition sets for each s_i with respect to each language L_j are precomputed (call them T_{ij}) and stored with each s_i.

We are given a regular expression pattern and derive a language L from it. We then try to find a homomorphism h_j such that $h_j(L) = L_j$ for some j. If we find one, we check each of the T_{ij} (simultaneously). As in the previous section, four cases occur. However, those

cases of Type 2 indicate that the search pattern is not present in those string segments, and they play no part in further searches. The best homomorphism to choose is one that does not conflate any of the terminals mentioned in the regular expression.

Suppose that the fraction of segments that are eliminated by this test is a. Then after a constant parallel time check of the precomputed transition sets, there remain $(1-a)p$ segments in which the search pattern might occur. Subdividing these and allocating them across all available processors gives p segments, each of length $(1-a)n$ to be searched using the algorithm described in the previous section. The overall time complexity of the optimised algorithm is

$$t_p = 1 + (1-a)\frac{n}{p} + \log p$$

For small values of p, making a large reduces the time complexity significantly, almost linearly in the fraction of segments eliminated.

The reason for calling the languages whose transition sets are precomputed indexing languages can now be seen. A traditional, word-based index corresponds to choosing an indexing language for each word – for example, the language L_{rat} generated by the regular expression x^*ratx^* – and a homomorphism that maps all other alphabet symbols to x. Using "rat" in an index corresponds to precomputing the transition sets for each segment relative to this language; which in turn amounts to labelling each segment according to whether it contains an instance of "rat", does not contain an instance, or contains a part of the string "rat". (Note that if we insist that text strings are broken at word boundaries, the last case does not arise.)

However, we are free to choose more complex indexing languages. Of particular interest in structured text search are languages that conflate parts of the document structure while emphasising other parts. Such languages help to speed up searches based on patterns involving structure as well as content.

In a parallel implementation, which we have been assuming, each T_{ij} is stored in processor i. However, in a sequential implementation, for which this extension to indexing is still useful, the T_{ij} for each j could be stored as a bit vector or as a list of segments in which the pattern might appear – much like a standard index.

There is an obvious connection between the these tables of bits and signature files. The major differences are:

- Data capturing some property of the text is based on segments rather than documents;
- Many different single-bit values are stored, one per indexing language, rather than a

single multiple-bit (perhaps 1024 bit) signature.

- Indexing languages can capture arbitrary patterns, while signatures capture information about words.

4 Examples

Documents defined in structured ways contain tags that delimit semantically interesting segments of the text. Searches that are aware of such tags are more powerful than searches based on content alone. For example, an occurrence of a phrase in a section heading, a name in an author field of a reference, a name in a mathematical formula all need tag information if they are to be separated from occurrences of the same string in other contexts.

Most documents have a well-defined structure. For example, it is natural to represent a tagged document as a tree structure, in which chapter tags and their associated text are close to the root of the tree, section heading tags and their text are below them, paragraphs still lower, with sentences or perhaps even words at the leaves. It seems plausible that searches on structure are more likely to involve patterns referring to tags at about the same level in the tree (sections within chapters) than to tags at widely differing levels (sentences within chapters). Thus interesting languages are those in which, for example, tags for high-level structure have been mapped to one generic tag, while those for low-level structure have been deleted entirely (recall that homomorphisms can erase, that is map terminals to the empty string). Similarly, it might be of interest to preserve low-level tags but delete high-level ones; or delete all tags enclosing special environments such as examples or mathematics. This directs interest to regions of the document in which appropriate kinds of tags appear.

Example 1 Consider the following text string, divided across five processors

$$\overbrace{a\ b\ c\ d}\ \overbrace{<\ e\ f\ g}\ \overbrace{>\ h\ i\ j}\ \overbrace{\{\ k\ l\ m}\ \overbrace{n\ o\ \}\ p}$$

Let us suppose an indexing language that preserves the bracket structure of the angle brackets, while removing the brackets structure of the curly braces (one might represent a chapter tag and the other a math tag). So L_1 is

$$L_1 = x^*(< x^* >)^* x^*$$

with homomorphism

$$\begin{aligned} h_1(?) &= x \\ h_1(<) &= < \end{aligned}$$

$$h_1(>) \;=\; >$$

The precomputed transition sets for the text string relative to this language are:

$$\overbrace{a\ b\ c\ d}^{\text{no}}\quad \overbrace{<\ e\ f\ g}^{\text{front}}\quad \overbrace{>\ h\ i\ j}^{\text{back}}\quad \overbrace{\{\ k\ l\ m}^{\text{no}}\quad \overbrace{n\ o\ \}\ p}^{\text{no}}$$

where the labels are shorthand for transition sets that do not contain an instance of the pattern (Type 2), or contain the front (Type 3) or back (Type 4) of a pattern. No segment contains the whole pattern in this example.

Now consider a search during normal operation for the pattern

$$< [\ ]^* e [\ ]^* >$$

The image of the language generated by this pattern under homomorphism h_1 is the language L_1 and so we check the precomputed transition sets. We immediately conclude that the only place in which this pattern may appear in the text string is in segments 2 and 3, which are searched using the ordinary algorithm. The value of a in this case is 0.6.

Notice that this example resembles the use of zones [5].

Similarly, languages in which letters infrequent in English text have been coalesced to a single letter could be chosen, since search patterns will tend not to include them.

Example 2 Consider the following text string, divided across six processors

$$\overbrace{a\ b\ c\ d}\quad \overbrace{e\ f\ g\ h}\quad \overbrace{i\ j\ k\ l}\quad \overbrace{e\ t\ e\ a}\quad \overbrace{t\ e\ e\ t}\quad \overbrace{m\ n\ o\ p}$$

An indexing language that preserves only common letters might be L_2

$$L_2 = (x^*(e+t)x^*)^*$$

with homomorphism

$$\begin{aligned} h_2(?) &= x \\ h_2(e) &= e \\ h_2(t) &= t \end{aligned}$$

The precomputed transition sets for the text string relative to this language are:

$\overbrace{a\ b\ c\ d}$	$\overbrace{e\ f\ g\ h}$	$\overbrace{i\ j\ k\ l}$	$\overbrace{e\ t\ e\ a}$	$\overbrace{t\ e\ e\ t}$	$\overbrace{m\ n\ o\ p}$
no	no	no	yes	yes	no

Now consider a search for the pattern

$$\Sigma^* tet \Sigma^*$$

The image of the language generated by this pattern under homomorphism h_2 is the language L_2 and so we check the precomputed transition sets. The only segment in which this pattern may appear in the text string is in segments 4 and 5, which are searched using the ordinary algorithm.

5 Conclusions

We have described an extension to indexing based on classes of languages, chosen to be homomorphic images of the languages generated by common search patterns. Such languages play the role of indexes with the added capability to search for patterns other than simple strings in the text. In parallel implementations of search, text strings are likely to be segmented across processors. The transition sets for each segment can be computed for each of the indexing languages in advance and stored with the segment. When a search pattern is presented, a homomorphism from it to one of the indexing languages is constructed if possible. A constant parallel time test then selects those segments in which an instance of the pattern cannot lie, with the expectation that in practice this will eliminate a large proportion of the text string. The speed-up achieved is almost linear in the fraction of text string segments eliminated. This technique makes it practical to use moderate parallelism in string search. It is more expressive than signature files because it allows patterns involving both contents (words) and structure (tags) to form part of the search query.

The technique has been presented for linear strings, but it generalises immediately to text stored as trees, in which regular expression search can also be carried out in logarithmic parallel time [3].

References

[1] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents

and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2:267–288, April 1984.

[2] Charles N. Fischer. *On Parsing Context-Free Languages in Parallel Environments*. PhD thesis, Cornell University, 1975.

[3] J. Gibbons, W. Cai, and D.B. Skillicorn. Efficient parallel algorithms for tree accumulations. *Science of Computer Programming*, 23:1–14, 1994.

[4] W. Daniel Hillis and G.L. Steele. Data parallel algorithms. *Communications of the ACM*, 29, No.12:1170–1183, December 1986.

[5] Fulcrum Technologies Inc. Ful/text reference manual. Fulcrum Technologies, Ottawa, Ontario, Canada, 1986.

[6] D.B. Skillicorn. Structured parallel computation in structured documents. Technical Report 95-379, Queen's University, Department of Computing and Information Science, March 1995.