Parallel Computing 40 (2014) 471–495

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Distributed text search using suffix arrays

Diego Arroyuelo [a,d,*,1], Carolina Bonacic [b,2], Veronica Gil-Costa [c,d,3], Mauricio Marin [b,d,f,4], Gonzalo Navarro [f,e,5]

[a] Dept. of Informatics, Univ. Técnica F. Santa María, Chile
[b] Dept. of Informatics, University of Santiago, Chile
[c] CONICET, University of San Luis, Argentina
[d] Yahoo! Labs Santiago, Chile
[e] Dept. of Computer Science, University of Chile, Chile
[f] Center of Biotechnology and Bioengineering, University of Chile, Chile

ARTICLE INFO

Article history:
Received 14 August 2013
Received in revised form 10 June 2014
Accepted 28 June 2014
Available online 11 July 2014

Keywords:
Distributed text search
Suffix arrays
Distributed text search engines

ABSTRACT

Text search is a classical problem in Computer Science, with many data-intensive applications. For this problem, *suffix arrays* are among the most widely known and used data structures, enabling fast searches for phrases, terms, substrings and regular expressions in large texts. Potential application domains for these operations include large-scale search services, such as Web search engines, where it is necessary to efficiently process intensive-traffic streams of on-line queries. This paper proposes strategies to enable such services by means of suffix arrays. We introduce techniques for deploying suffix arrays on clusters of distributed-memory processors and then study the processing of multiple queries on the distributed data structure. Even though the cost of individual search operations in sequential (non-distributed) suffix arrays is low in practice, the problem of processing multiple queries on distributed-memory systems, so that hardware resources are used efficiently, is relevant to services aimed at achieving high query throughput at low operational costs. Our theoretical and experimental performance studies show that our proposals are suitable solutions for building efficient and scalable on-line search services based on suffix arrays.

1. Introduction

In the last decade, the design of efficient data structures and algorithms for textual databases and related applications has received a great deal of attention, due to the rapid growth of the amount of text data available from different sources. Typical applications support text searches over big text collections in a client–server fashion, where the user queries are answered by a dedicated server [15]. The server efficiency—in terms of running time—is of paramount importance in cases where the

* Corresponding author. Address: Av. España 1680, Valparaíso, Chile. Tel.: +56 2 432 6722; fax: +56 2 432 6702.
E-mail addresses: darroyue@inf.utfsm.cl (D. Arroyuelo), carolina.bonacic@usach.cl (C. Bonacic), gvcosta@unsl.edu.ar (V. Gil-Costa), mauricio.marin@usach.cl (M. Marin), gnavarro@dcc.uchile.cl (G. Navarro).

[1] Funded by FONDECYT Grant 11121556, Chile.
[2] Funded in part by DICYT-USACH Grant 061319BC.
[3] Funded in part by CONICET-UNSL Grant 30310.
[4] Funded in part by FONDEF IDeA Grant CA12i10314 and Basal funds FB0001, Conicyt, Chile.
[5] Funded with Basal funds FB0001, Conicyt, Chile.

http://dx.doi.org/10.1016/j.parco.2014.06.007
0167-8191/

services demanded by clients generate a heavy work load. A feasible way to overcome the limitations of sequential computers is to resort to the use of several computers, or processors, which work together to serve the ever increasing client demands [19].

One such approach to efficient parallelization is to distribute the data onto the processors, in such a way that it becomes feasible to exploit locality via parallel processing of user requests, each on a subset of the data. As opposed to *shared-memory* models, this *distributed-memory* model provides the benefit of better scalability [44]. However, it introduces new problems related to the communication and synchronization of processors and their load balance.

This paper studies the parallelization of text indexes, in particular *suffix arrays* [39], in distributed memory systems, and describes strategies to reduce the inter-processor communication and to improve the load balance at search time.

1.1. Indexed text searching

The advent of powerful processors and cheap storage has enabled alternative models for information retrieval, other than the traditional one of a collection of documents indexed by a fixed set of keywords. One is the *full text* model, in which the user expresses its information need via words, phrases or patterns to be matched for, and the information system retrieves those documents containing the user-specified patterns. While the cost of full-text searching is usually high, the model is powerful, requires no structure in the text, and is conceptually simple [5].

To reduce the cost of searching a text, specialized indexing structures are adopted. The most popular are *inverted indexes* [5,10,68]. Inverted indexes are efficient because their search strategy is based on the vocabulary (the set of distinct words in the text), which is usually much smaller than the text, and thus fits in main memory. For each word, the list of all its occurrences (positions) in the text is stored. Those lists are large and may be stored on secondary storage, or in the main memory of the cluster nodes [10]. However, inverted indexes are suitable only for searching natural-language texts (which have clearly separated words that follow some convenient statistical rules [5]). *Suffix arrays* or PAT *arrays* [39,28], on the other hand, are more sophisticated indexing structures, which are superior to inverted indexes when searching for phrases or complex queries such as regular expressions [5]. In addition, suffix arrays can be used to index texts other than natural language. Examples of these applications include computational biology (DNA or protein sequences), music retrieval (MIDI or audio files), East Asian languages (Chinese, Korean, and others), and other multimedia data files.

Pattern search on suffix arrays is based on binary search [39,28]; see Section 2.1 for further details. Processing a single query X of length m in a text of length n takes $O(m \log n)$ time on the standard sequential suffix array. One can also achieve $O(m + \log n)$ time, yet by storing an extra array that doubles the space usage. Hence, trying to reduce such query time by using a distributed-memory parallel computer of P processors is not very attractive in practical terms. In real applications, however, many queries arrive at the server per unit of time. Such work load can be served by taking batches of Q queries. Processing batches in parallel is appealing in this context, as one is more interested in improving the throughput of the whole process rather than improving single operations.

To achieve this goal, a pragmatic (though naive) strategy would be to keep a copy of both the whole text database and the search index in each server machine and route the queries uniformly at random among the P machines. For very large databases, however, each machine is forced to keep a copy of a large suffix array, often in secondary memory, which can drastically degrade performance. A more sensible approach is to keep a single copy of the suffix array evenly distributed over the P main memories. Now the challenge is to achieve efficient performance on a cluster of P machines that must communicate and synchronize in order to serve every batch of queries. This is not trivial: on a naive partitioning of the suffix array, most array positions are expected to reference to text stored in a remote memory. We study these problems in this paper in order to achieve efficient text searching.

1.2. Problem definition and model of computation

In its most basic form, the *full-text search* problem is defined as follows: Given a *text* $T[1 \dots n]$, which is a sequence of n symbols from an ordered alphabet $\Sigma = \{1, \dots, \sigma\}$, and given a *search pattern* $X[1 \dots m]$ (also over Σ), we want to find all the occurrences of X in T. There are different kinds of queries for full-text search, depending on the application:

- locate queries, where one wants to report the starting positions in the text of the pattern occurrences, that is, the set $O = \{i | 1 \leqslant i \leqslant n - m + 1 \wedge T[i \dots i + m - 1] = X[1 \dots m]\}$.
- count queries, where one wants to count the number of pattern occurrences, that is, compute $|O|$.
- exist queries, where one wants to check whether X occurs in T or not, that is, we want to determine whether $O = \emptyset$ or not.

In this paper we will study a variant of this problem suitable for parallel processing when throughput is emphasized: given a set of Q patterns $\{X_1[1 \dots m], X_2[1 \dots m], \dots, X_Q[1 \dots m]\}$, one wants to find the occurrences of each of these strings in T (we use equal length for all the patterns for simplicity, but our results do not depend on that). We will focus on count queries. In such a case, one wants to obtain the number of occurrences of each of the Q search patterns. The reason is that, on suffix arrays, the algorithmic complexity of the problem lies on the counting part, and when this is solved, locating reduces to merely listing the values in an array range. In addition, as we discuss in the Conclusions, it turns out that the strategies that are best for counting are also the most promising ones for locating. On the other hand, many relevant applications rely solely