Suffix Arrays in Parallel*

Mauricio Marín[1] and Gonzalo Navarro[2]

[1] University of Magallanes, mmarin@ona.fi.umag.cl
[2] University of Chile, Center for Web Research (www.cwr.cl)
gnavarro@dcc.uchile.cl.

Abstract. Suffix arrays are powerful data structures for text indexing. In this paper we present parallel algorithms devised to increase throughput of suffix arrays on a multiple-query setting. Experimental results show that efficient performance is indeed feasible in this strongly sequential and very poor locality data structure.

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 Web [1]. To reduce the cost of searching a full large text collection, specialized indexing structures are adopted. *Suffix arrays* or PAT *arrays* [1] are examples of such index structures. They are more suitable than the popular inverted lists for searching phrases or complex queries composed of regular expressions [1]. A fact to consider in parallel processing natural language texts is that words are not uniformly distributed both in the text itself and the queries provided by the users of the system. This produces load imbalance.

The efficient construction in parallel of suffix arrays has been investigated in [3,2]. In this paper we focus on query processing. We propose efficient parallel algorithms for (1) processing queries grouped in batches of Q queries, and (2) load balancing properly this process when dealing with biased collections of words such as in natural language.

2 Global Suffix Arrays in Parallel

The suffix array is a binary search based strategy. The array contains pointers to the document terms, where pointers identify both documents and positions of terms within them. The array is sorted in lexicographical order by terms. A search is conducted by direct comparison of the terms pointed to by the array elements. A typical query is finding all text positions where a given substring appears in and it can be solved by performing two searches that locate the delimiting positions of the array for the substring.

We assume a broker (server) operating upon a set of P machines running the BSP model [4]. The broker services clients' requests by distributing queries onto

* Funded by Millenium Nucleus Center for Web Research, Grant P01-029-F, Mideplan, Chile.

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 338–341, 2003.

the P machines implementing the BSP server. We assume that under a situation of heavy traffic the server processes batches of $Q = q\,P$ queries.

A suffix array can be distributed onto the processors using a *global index* approach in which a single array of N elements is built from the whole text collection and mapped evenly on the processors. In this case, each processor stands for an interval or range of suffixes (lexicographic partition). The broker machine mantains information of the values limiting the intervals in each machine and route queries to the processors accordingly. This fact can be the source of load imbalance in the processors when queries tend to be dinamically biased to particular intervals. We call this strategy G0.

In the *local index* strategy, on the other hand, a suffix array is constructed in each processor by considering only the subset of text stored in its respective processor. Unlike the global approach, no references to text postitions stored in other processors are made. However, for every query it is necessary to search in all of the processors in order to find the pieces of local arrays that form the solution for a given interval defined by the query. It is also necessary to broadcast every query to every processor. We have found both theoretically and experimentally that the global index approach offers the potential of better performance.

A drawback of the global index approach is related to the possibility of load imbalance coming from large and sustained sequences of queries being routed to the same processor. The best way to avoid particular preferences for a given processor is to send queries uniformly at random among the processors. We propose to achieve this effect by multiplexing each interval defined by the original global array so that if the array element i is stored in processor p, then the elements $i+1$, $i+2$, ... are stored stored in processors $p+1$, $p+2$, ... respectively in a circular manner. We call this strategy G1.

In this case, any binary search can start at any processor. Once a search has determined that the given term must be located between two consecutive entries k and $k+1$ of the array in a processor, the search is continued in the next processor and so on, where at each processor it is only necessary to look at entry k of the local arrays. In general, for large P, the inter-processors search can be done in at most $\log P$ additional BSP supersteps by performing a binary search accross processors.

The binary search on the global index approach can lead to a certain number of accesses to remote memory. A very effective way to reduce this is to associate with every array entry the first t characters of the terms respectively. The value of t depends of the average length of terms. This reduced remote accesses to less than 5% in [2], and less than 1% in our experiments, for relatively small t.

In G0 we keep in each processor an array of P strings of size ℓ marking the delimiting points of each interval of G0. The broker machine routes queries uniformly at random to the P real processors, and in every processor a $\log P$ time binary search is performed to determine to which processor send a given query (we do so to avoid the broker becoming a bottleneck). Once a query has been sent to its target processor it cannot migrate to other processors as in the case of G1. That is, this strategy avoids the inter-processors $\log P$ binary search.

3 Experimental Results and Conclusions

We now compare the multiplexed strategy (G1) with the plain global suffix array (G0). For each element of the array we keep t characters which are the t-sized prefix of the suffix pointed to by the array element. We found $t = 4$ to be a good value for our text collection. We also implemented the local index strategy, but it was always at least 3 times slower than G0 or G1.

The text collection is formed by a 1GB sample of the Chilean Web retrieved by the search engine www.todocl.cl. We treated it as a single string of characters, and queries were formed by selecting positions at random within this string. For each position a substring of size 16 is used as a query. In the first set of experiments these positions were selected uniformly at random. Thus load balance is expected to be near optimal. In the second set of experiments we selected at random only the positions whose starting word character were one of the four most popular letters of the Spanish language. This produces large imbalance as searches tend to end up in a subset of the global array. Experiments with an actual set of queries from users of "todocl" produced results between the uniform and biased cases.

The experiments were performed on a PC cluster of 16 machines. Runs with more than 16 processors were performed by simulating virtual processors. At each superstep we introduced $1024/P$ new queries in each processor. Most speedups obtained against a sequential realization of suffix arrays were super-linear. This was not a surprise since due to hardware limitations we had to keep large pieces of the suffix array in secondary memory whilst communication among machines was composed by a comparatively small number of strings.

The whole text was kept on disk so that once the first t chars of a query were found to be equal to the t chars kept in the respective array element, a disk access was necessary to verify that the string forming the query was effectively found at that position. With high probability this required an access to a disk file located in other processor, case in which the whole query is sent to that processor to be compared with the text retrieved from the remote disk.

Though we present running time comparisons below, what we considered more relevant to the scope of this paper is an implementation and hardware independent comparison between G0 and G1. This came in the form of two performance metrics devised to evaluate load balance in computation and communication. They are the average maximum across supersteps. During the processing of a query each strategy performs the same kind of operations, so for the case of computation the number of these ones executed in each processor per superstep suffices as an indicator of load balance for computation. For communication we measured the amount of data sent to and received from at each processor in every superstep. We also measured balance of disk accesses.

Table 1.1 shows results for queries biased to the 4 popular letters. Columns 2, 3, and 4 show the ratio G1/G0 for each of the above defined performance metrics (average maximum for computation, communication and disk access). These results confirm intuition, that is G0 can degenerate into a very poor performance strategy whereas G1 is a much better alternative. G1 is independent

of the application but, though well-balanced, it tends to generate more message traffic due to the inter-processors binary searches. The differences among G1 and G0 are not significant for the case of queries selected uniformly at random. G1 tends to have a slightly better load balance.

Table 1. G1/G0 ratios.

P	comp	comm	disk	P	comp	comm	disk
2	0.95	0.90	0.89	16	0.39	0.35	0.36
4	0.49	0.61	0.69	32	0.38	0.29	0.24
8	0.43	0.45	0.53	64	0.35	0.27	0.17

(1) Performance metrics

P	Biased	Uniform
4	0.68	0.78
8	0.55	0.78
16	0.61	0.86

(2) Running times

As speed-ups were superlinear due to disk activity, we performed experiments with a reduced text database. We used a sample of 1MB per processor which reduces very significantly the computation costs and thereby it makes much more relevant the communication and synchronization costs in the overall running time. We observed an average speed-up efficiency of 0.65.

Table 1.2 shows running time ratios obtained with our 16 machines cluster. The biased workload increased running times by a factor of 1.7 approximately. In both cases G1 outperformed G0, but G1 loses efficiency as the number of processors increases. This is because, as P grows, the effect of the inter-processor binary searches becomes more significant in this very low-cost computation scenario.

Note that the above G0 strategy can be extended to approximate G1 by partitioning the array in $V = k\,P$ virtual processors and mapping the V pieces of the array in a circular manner on the P actual processors. Preliminary experiments show that this strategy tends to significantly reduce the imbalance of G0 at a small k value.

Yet another method which solves both load imbalance and remote references is to re-order the original global array so that every element of it contains only pointers to local text (or most of them). This becomes similar to the local index strategy whilst it still keep global information which avoids the P parallel binary searches and broadcast per query. Unfortunately we now lose the capability of performing the inter-processors $\log P$-cost binary search. We are currently investigating ways of performing this search efficiently using little extra space.

References

1. R. Baeza and B. Ribeiro. *Modern Information Retrieval.* Addison-Wesley., 1999.
2. J. Kitajima and G. Navarro. A fast distributed suffix array generation algorithm. In *6th Symp. String Processing and Information Retrieval*, pages 97–104, 1999.
3. G. Navarro, J. Kitajima, B. Ribeiro, and N. Ziviani. Distributed generation of suffix arrays. In *8th Symp. Combinatorial Pattern Matching*, pages 102–115, 1997. LNCS 1264.
4. L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.