Abstract

Searching for arbitrary combinations of words in large, unformatted databases containing natural language documents is an important application. In this paper we describe the present status of a parallel free-text search system on the Connection Machine[1] that uses an indexing scheme.

1 Introduction

A free-text search system allows the user to search for an arbitrary combination of words in a database of natural language documents. Generally, such databases tend to be extremely large. The computational requirements in searching such a database are often unrealistic when conventional computers are used. Several attempts in using parallel machines have been described in the literature. Stanfill and Kahle[1] described such a system on the Connection Machine, which has a SIMD architecture. Their system uses a scheme called *surrogate coding* to represent the documents. Each processor maintains one or more *surrogate table(s)*. A surrogate table is a binary array with a specific length k. To insert a word into the database, a fixed number (say i) of hash codes are generated and the surrogate table bits in positions corresponding to these codes are set. Searching for a word is simply, generating all k hash codes for the word and checking to see if **all** the corresponding bits in the surrogate table are set. Figure 1 illustrates this scheme. Stanfill and Kahle show that with $i = 10$ and $k = 512$, up to 30 words could be stored in a surrogate table with 0.03% probability of making a false hit.

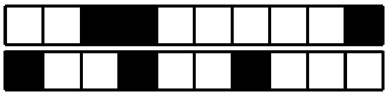

Figure 1. The surrogate table scheme ($k = 10, i = 3$), if both the words were entered into the same surrogate table, the search for a new word that hashes to, say, (0,2,3) will produce a false match.

The entire database is distributed over the processor array in this form. A query is processed word by word, with every processor searching its surrogate tables to see if the word is present. If the database is bigger than the holding capacity of the processor array, several overlays need to be made.

Stone's analysis[2] of this system illustrated how an efficient sequential algorithm using indexing can match the performance of a parallel algorithm that doesn't use indexing. Stone's contention was that the

[1] Connection Machine is a registered trademark of Thinking Machines Corporation

use of indexing dramatically reduces the search space and I/O requirements. Thus parallelism per se would not yield a faster solution if the abundance or processing power is wasted in searching portions of the database which could have been eliminated en masse by the use of indexing. Though Stone's reasoning is not universally relevant, in the case of free-text search it does remain valid. Stone suggested that a parallel algorithm that uses indexing may perform better, provided that the inter-processor communication involved could be carried out efficiently.

2 Description of the system

In this paper, we describe the present status of a parallel free-text search system, on the Connection Machine, that uses indexing. The Connection Machine is a parallel computer system with a SIMD architecture, comprising up to 65, 536 bit-serial processing elements. Each processor has up to 65, 536 bits of local memory. Each group of 16 processors is equipped with a router node which provides highly optimized interprocessor communication. The architecture of the Connection Machine is described in [3] and [4]. The *DataVault*[2] is a data-parallel file system for the Connection Machine. Instead of having a single stream of bits, a DataVault file has many streams of bits, one stream per selected processor. The actual storage consists of 39 disk drives operating in parallel. The sequencing of data is transparent to the programmer. The DataVault is described in detail in [5].

2.1 Data structures

We use a hashed-indexing scheme to represent the database. Each word is mapped to a hash-table location, where a list of couplets is maintained. Each couplet consists of a word that mapped to the location and a pointer to the place in a parallel file, beginning from which all the document numbers containing the word are kept in ascending order. The database then consists of the parallel file of document numbers and the hash-table.

2.2 Algorithm for query processing

Processing a query is as follows:

- *Step 1* Assign each query word to a cluster of processors. Each cluster, which is a sub-hypercube, has enough processors to han-

[2] DataVault is a registered trademark of Thinking Machines Corporation

dle the longest list of documents that may be retrieved from the database.

- *Step 2* Each cluster hashes its word, enters the appropriate index list in the hash-table and determines the pointer to the parallel file location from which the list of documents containing the word begins.
- *Step 3* Each cluster loads the corresponding list of document numbers so that every entry gets assigned to a separate processor in the cluster.
- *Step 4* Combine the lists of document numbers by performing a modified bitonic sort over all clusters.
- *Step 5* Reduce the sorted list of documents such that multiple occurrences of document numbers are removed after appropriate scoring.
- *Step 6* Sort the document numbers again, according to the associated scores.
- *Step 7* Report the numbers of the documents with the highest scores.

Modified bitonic sort, mentioned in step 4, takes n lists, each sorted in ascending order, and combines them to produce a single sorted list. This is straightforward case of the general bitonic sort. In every step of the sort, two neighboring lists are merged together. After $\lg n$ steps, the resulting list will be completely sorted.

2.2.1 Analysis

Let there be n words in the query and m be the average length of the list of documents containing a query word. The bitonic merge of two lists of length l takes $O(\lg l)$ In the first step, lists of length l are merged to yield $n/2$ lists of length $2m$. Thus, the total time taken for the sort is: $\lg m + \lg 2m + \lg 4m .... + \lg \frac{n}{2}m$, which reduces to $\lg m \lg n + \frac{(\lg \frac{n}{2} - 1)\lg \frac{n}{2}}{2}$. The asymptotic time complexity is therefore $O(\lg m \lg n) + O(\lg^2 n)$. The constant of the second term is lower compared to that of the first.

2.3 Installing and deleting documents

Installing a document is done similarly in parallel. Each word in the document are assigned to a different cluster of processors. Each cluster of processors now updates the hash-table and adds the current document number to the list in the parallel file. Deleting a document is similar.

The actual document resides in a front-end secondary storage device. Before installing a document it is pruned of all words that are deemed too common.

2.4 Effect of indexing - A comparison

In this section we compare the performance of this algorithm with the algorithm by Stanfill and Kahle which does not use indexing and the sequential query processing algorithm which uses indexing. We borrow the models used by Stone[2] for this comparison. The parameters used in the comparison are as listed in Table 1. (Note : In all schemes the database has one entry per for every word-document pair)

Parameter	*Description*	*Value*
b	No. of bits per entry(SIMD unindexed)	2^5
b'	No. of bits per entry(serial)	2^5
b''	No. of bits per entry(SIMD indexed)	2^5
D	No. of documents in the database	2^{25}
L	No. of distinct words in the language	2^{16}
P	No. of processors in the SIMD machine	2^{16}
Q	No. of words in a query	2^5 to 2^{15}
S	Memory (in bits) in one SIMD processor	2^{16}
W	Number of distinct words per document	2^7

Table 1. Summary of Notation

Stone identifies five quantities for comparison.

I/O traffic : The average length of the document list corresponding to any word is WD/L. Therefore, the total amount of data-transfer for a query is $Qb''WD/L$. With the typical parameter values in Table 1, this would evaluate to 2^{26} to 2^{36}

Number of I/O operations : Each processor can read/write one document number during an I/O operation. Therefore, the number of I/O operations involved are QWD/PL. For the typical values assumed, this computes to 2^5 to 2^{15}. However, the number of I/O operations does not constitute a meaningful comparison because the amount of data transferred in one I/O operation is not the same for the serial and parallel algorithms.

Number of Searches : The search involved is to determine the location of the list of document numbers from the hash table. This needs to be performed once for every query word. But a number of query words can be handled in parallel. Since the average length of each list of documents is $b''WD/L$, and the total memory in the CM processor array is PS, the number of query words that can be handled in parallel is $PSL/b''WD$. With the typical values, this figure ranges from 1 to 2^{11}.

Number of combines and selects : The combine and select operations involved here are different from those of the other two algorithms. We can view the sorting as a combine 0 and the subsequent reduction as a select. Since the average list length is WD/L, the number of combine operations is $\lg \frac{WD}{L} \lg Q + \frac{lg(\frac{Q}{2}-1)\lg\frac{Q}{2}}{2}$. For the typical parameters, this evaluates to a maximum of 90. The final reduction is based on collapsing the duplicate document numbers and sorting them according to their scores. The former is a parallel-prefix operation which would take $O(\lg \frac{QWD}{L})$ time and the latter would take $O(lg^2 \frac{QWD}{L})$ time. Thus approximately this last step would take maximum values around 1000.

The comparison of the three algorithms is presented in Table 2. The expressions and figures for the other two algorithms were derived in [2].

Quantity Estimated	*SIMD (w/o indexing)*	*Serial*	*SIMD (with indexing)*
I/O traffic (bits)	bWD 2^{37}	$Q(b'WD)/L$ 2^{26} to 2^{36}	$Q(b''WD)/L$ 2^{26} to 2^{36}
I/O operations	bWD/PS 2^{10}	Q 2^{5} to 2^{15}	QWD/PL 2^{5} to 2^{15}
Searches	$QbWD/PS$ 2^{15} to 2^{25}	Q 2^{5} to 2^{15}	$PSL/b''WD$ 1 to 2^{11}
Combines	$QbWD/PS$ 2^{15} to 2^{25}	QWD/L 2^{21} to 2^{31}	$O(\lg \frac{WD}{L} \lg Q)$ $+O(\lg^2 Q)$ 90
Selects	bWD/PS 2^{10}	D 2^{25}	$O(\lg^2 \frac{QWD}{L})$ 1000

Table 2. Comparison of Algorithms

3 Performance

A first version of the system has been built and tested. The results of running queries on a database of 512 documents, each consisting of about 128 words on the average, is shown in Table 3. The system uses 16k processors configured as a hypercube on the Connection Machine model CM-2 with number of virtual processors to number of physical processors ratio (V/P ratio) of 1. The size of the database is limited by a bug in the DataVault software which forces the data in all the processors to be written out to the DataVault file even when only a subset of processors is selected. When DataVault I/O on selected processors becomes possible we estimate that 32,000 documents could be stored without degrading performance. For databases with more documents, the V/P ratio need to be larger than unity, which in turn would result in a performance degradation by a constant factor.

The Modified bitonic sort was coded in C*. It makes use of the fact that the lists pulled out of the DataVault are already sorted and therefore can be viewed as a set of bitonic sequences when taken two at a time. If the average list length is m and the number of words in the query is n, this algorithm should take $O(\lg m \lg n) + O(\lg^2 n)$time. In this implementation, since the average list length was a constant, the time complexity was $O(\lg^2 n)$, as is evident from Table 3. The fast low-level sort uses the low-level Parallel Instruction Set (PARIS) calls for ranking and sending. The former algorithm, if implemented in micro-code, should take less time than the second.

According to Salton and Buckley[6], a parallel search system on the CM which does not use indexing typically takes 0.04 seconds for matching a 200-word query with the documents in a database of 16,000 documents and a further 0.02 seconds for post-processing. Since the two systems use fundamentally different approaches, it is not logical to compare the performance of parts of the system. As seen in the analysis in the previous section, the total amount of information transferred between the stored database and the CM processor array is likely to be less in our system(Step 3). This is because only the relevant documents are loaded onto the CM processor array. Further, our system will not fail to detect a match or detect spurious matches because the representation of each document is unique and complete. This is in contrast with the other system, in which the signature based representation can potentially cause false hits. Both systems will suffer performance degradation by a constant factor when the size of the database exceeds a certain threshold, thereby making overlays necessary. But our threshold is likely to be higher.

The I/O support for the DataVault is not complete yet. Furthermore some serious bugs still remain. For instance, Step 3 cannot be executed in parallel at present. As mentioned earlier, the size of the database is limited by a bug in the DataVault software. It is hoped that these shortcomings will be remedied in the newer releases of DataVault system software. With that, the performance of our system will be better.

Query size (words)	*CM time (Seconds)*		
	Sorting		Post-processing
	Modified Bitonic	Fast low-level	
8	0.29	0.05	0.06
10	0.29	0.05	0.06
16	0.40	0.05	0.06
20	0.43	0.05	0.06
30	0.44	0.05	0.06
32	0.52	0.05	0.06
40	0.61	0.05	0.06
50	0.54	0.05	0.06
60	0.56	0.05	0.06
64	0.65	0.05	0.06

Table 3. Performance results

4 Conclusion

In this paper we presented a parallel free-text search system that uses indexing. The system promises to be efficient when the database is very large. The use of indexing resulted in the reduction of the search space thereby making the performance of the system comparable to the fastest uni-processor systems.

References

[1] Stanfill, Craig & Kahle, Brewster. 'Parallel Free-text Search on the Connection Machine System', *CACM* , Dec 1986 Vol. 29 #12 1229-1239

[2] Stone, Harold. 'Parallel Querying of Databases: A Case Study', *IEEE Computer*, Oct 1987 11-21

[3] Hillis, Daniel. *The Connection Machine*, MIT Press Cambridge, Mass. 1985

[4] Thinking Machines Corporation. *C* Reference Manual,* Thinking Machines Corporation, Cambridge, Mass.

[5] Thinking Machines Corporation. *DataVault Reference Manual,* Thinking Machines Corporation, Cambridge, Mass.

[6] Salton, Gerald & Buckley, Chris. 'Parallel Text Search Methods', *CACM* , Feb 1988 Vol. 31 #2 202-215