Document Retrieval Performance On Parallel Systems

David Hawking
Cooperative Research Centre For Advanced Computational Systems
Department of Computer Science
The Australian National University
Canberra, ACT, AUSTRALIA
email: dave@cs.anu.edu.au

Abstract

The problem of efficiently retrieving and ranking documents from a huge collection according to their relevance to a research topic is addressed. A broad class of queries is defined and, based on previous work, a parallel system architecture capable of handling them is proposed. The time cost of the steps involved in query processing is analysed and the space requirements of the data structures used are outlined. The result is a model, characterised by parameters which can be derived from machine configuration information and some simple empirical measurements, from which collection capacities and likely query processing rates may be determined for given hardware configurations. The performance of a prototype implementation for a 128 node machine is analysed in terms of the model and conclusions are drawn on the relative importance of I/O and CPU parallelism.

Keywords: Document Retrieval, Text Processing, Parallel Algorithms

1 Introduction

Locating the documents in a large collection which best satisfy an information need is central to many modern professions. Automatic retrieval systems now deliver good results and are regarded as essential tools because manual searches are too time-consuming and too expensive. Projected growth in size of the largest collections of electronic text suggests that automatic systems capable of handling up to a terabyte of text will soon be required.

In a state-of-the-art automatic retrieval system, relevance scores are computed for each document based on the extent to which they match a query derived from the research topic or information need. Documents are then returned to the user in order of decreasing relevance score.

This paper assumes that suitable methods of query generation and relevance scoring are available and focuses on the algorithms and computing resources needed to efficiently process queries over collections approaching a terabyte in size. Users will demand that typical queries should be processed in just a few seconds.

In addition to query processing *per se* the larger problem of building, from the collection, the data structures necessary for fast query processing must also be considered. Finally, collections may be subject to change with time. Consequently, a design is needed which will avoid the necessity to completely rebuild data structures merely to add, remove or alter a small number of documents.

The magnitude of the problems of structure building and query processing over terabyte collections is clearly beyond the scope of current workstations or personal computers. It is expected that parallel systems will provide a solution.

The only known work addressing retrieval over huge text collections was carried out by Stanfill and his collaborators at Thinking Machines Corporation. Stanfill and Thau [11] describe parallel retrieval algorithms for the Connection Machine CM2 which they claim are capable of processing queries (not including proximity operations) over 8 terabytes of text. The emphasis of their paper (and also a related paper by Stanfill, Thau and Waltz [12]) is on the scoring and ranking of documents. They did not address the larger problem of storing and indexing multiple terabytes of data.

Assuming a distributed memory parallel machine, this paper will formally characterise query processing for a broad class of queries. Algorithms and data structures capable of solving the retrieval and structure building problems will be presented and their space and time requirements will be analysed in terms of the model. Experiments with a prototype implementation and a scaled-down document collection will be reported and used to explore the applicability of the model. Finally, the model, supplemented by empirical observations, will be used to suggest ways in which performance may be improved.

2 Approaches to Parallelising Document Retrieval

Most approaches to parallelising document retrieval divide the collection (or the collection of document signatures) across the processing nodes. This is true of the Thinking Machines Corporation work [11, 12], past work at the Australian National University [6, 5] and also the analyses by Stone [13], Salton and Buckley [9], Cringean et al. [2] and Skillicorn [10].

The system described by Reddaway [8] operates somewhat similarly. Inverted files are (logically) partitioned into two parts: a bit map recording which documents contain at least one occurrence of the term; and a record of positional information. The first phase of query processing consists of boolean operations on long bit maps. In effect, thousands of documents are processed in parallel.

Lofti-Jam and Kent [7] take an alternative approach in which query terms are distributed across processors. They report a bottleneck due to the processors loading the relevant parts of the inverted file from a single central source.

The former approach seems to offer the following advantages over the latter when dealing with huge collections:

1. Individual nodes need only access the local fraction of the collection leading to a significant reduction in number of bits required for pointers. This reduction may enable use of efficient machine operations in circumstances in which they would otherwise be inapplicable.

2. On systems where disk storage is local to the processing nodes, such as networks of workstations, no non-local disk accesses are required, with consequent reduction in network load.

3. Good load balance is easily achieved.

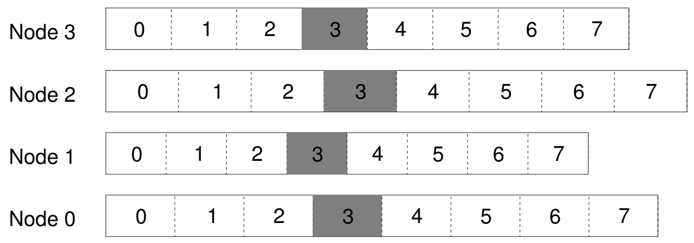


Figure 1: An illustrative collection divided across 4 nodes. The data held by a node is further divided into 8 approximately equal sized chunks. Note that there may be variation in chunk sizes and in the total amount of data held on the nodes. The set of similarly numbered chunks on the nodes is called a sub-collection. In this example, sub-collection 3 is shaded.

3 Formal Characterisation of the Problem

A document collection C comprises S_C characters, each represented in B_{char} bits using a uniform encoding scheme. The number of individual documents in C is N_d and the total number of words is N_w. The set of distinct words in the collection (the vocabulary) is V and the length of the longest word in the vocabulary is L_{max}.

A query Q comprises query terms $T_1, ...T_n$ which may be words, phrases or word prefixes. The relevance score R_d (representable in B_R bits) for a document is assigned on the basis of the frequencies of term occurrences and possibly on the basis of lexical distance separating occurrences of particular terms. This model of a query does not support arbitrary strings or regular expressions as terms but is typical of the query types supported by successful current systems. It supports boolean, vector-space, probabilistic and distance-based models.

Query Q is to be processed over collection C using a parallel (possibly distributed) system with N_n processing nodes. In addition to these nodes, there is a separate node (a front-end or separate workstation) which runs the user interface and broadcasts query commands to the processing nodes. Distributed memory is assumed.

The system is further assumed to be equipped with a high performance disk system either in the form of a disk array or in the form of local disks connected to some or all processing nodes. Total disk bandwidth is BW characters/sec. and the disk system is capable of I I/O operations/sec. on average. Total available disk space is S_{disk} characters.

Communication between nodes is necessary to determine global collection frequencies (if needed in relevance scoring) and to produce global relevance rankings. Time taken to calculate a global sum of frequency counts (t_{gsum}) and time to calculate a global maximum of relevance scores (t_{gmax}) are the key parameters here. Communication between the user interface and the processing nodes (required to broadcast queries and to return lists of relevant documents and the retrieved documents themselves) is assumed not to be a bottleneck.

4 Proposed Design

Faloutsos [3] surveys available methods for accessing text data. Of these alternatives, an inverted file method is chosen for the problem described here, taking into account the analyses of Stone [13] and Salton and Buckley [9]. Accordingly, the following approaches to data structure building and query processing are now proposed.

Approach to Data Structure Building

1. Distribute responsibility for the collection across the N_n processing nodes, taking care to achieve good load balance.

2. Further reduce the problem by breaking the data held on a node into N_p pieces which can be individually indexed. These pieces will be called *chunks*. Documents will not be split across chunks. Chunks will have a size of approximately S_{chunk} characters, chosen to provide acceptably rapid index rebuilding in the case of collection changes without overloading the filesystem with an unmanageable number of small files. Controlling the chunk size also limits the amount of primary memory required for efficient index building. Across the whole machine, the ratio of the largest chunk to the average will be denoted as r_{chunk}. Figure 1 illustrates the two-dimensional division of the data represented by steps 1 and 2.

3. For each chunk, construct an inverted file index and dictionary.

4. On each node, construct a super dictionary from the dictionaries for all of the chunks associated with the node.

Approach to Query Processing

Step 1 The user interface accepts a search query and broadcasts it to all nodes.

Step 2 Each node updates the appropriate document accumulators according to the chosen relevance scoring algorithm.

Step 3 The nodes combine to produce a merged list of relevant documents in order of decreasing score and pass this (or the first k items) back to the user interface. Items in the list are tagged in such a way as to permit efficient and unambiguous retrieval if requested by the user.

Step 4 When the user selects a document for viewing, the user interface directly requests it from the appropriate node.

Efficient support for query processing as outlined above will require the data structures listed below. In each case an attempt is made to identify the storage requirements across the whole machine for the uncompressed data structures.

Data Structures Required

Raw Data: Needed to enable viewing of retrieved documents. Space required is by definition $S_C \times B_{char}$ bits.

Document Table: Allows documents to be rapidly located on disk during relevance scoring and viewing by the user. Since each node is responsible for approximately S_C/N_n characters of data, $\log_2(S_C/N_n)$ bits are needed for each pointer. Total space required across the machine is thus $N_d \times \lceil \log_2(S_C/N_n) \rceil$ bits.

Document Accumulators: Record the relevance score for each particular document as it accumulates during query processing. Accumulators may be provided:

1. for every document; or
2. only for documents which achieve a non-zero score; or

3. only for the first N_{DA} documents to do so.

Assuming the first case, the storage required will be $N_d \times B_R$ bits.

Inverted File Indexes Record the location of term occurrences within a chunk to the character level (word level locations would save a small number of bits per entry). Assuming that index entries are byte offsets within the corresponding chunk, the number of bits required for each entry is determined by the number of characters in the largest chunk. The number of bits is given by $\lceil \log_2(r_{chunk} \times S_{chunk}) \rceil$. The total number of index entries across the whole machine is given by the number of words in the entire collection, and hence total space required is $\lceil N_w \times \log_2(r_{chunk} \times S_{chunk}) \rceil$.

Super Dictionary Lists all words in the collection and enables efficient term frequency discovery and fast index access. Each node's part of the super dictionary contains one record for each of the $|V_n|$ distinct words found in all of the chunks associated with node n. If the super dictionary is represented as a rectangular table, then each record will contain space for the maximum length word ($L_{max} \times B_{char}$ bits) plus an [offset, length] pair for each of the N_p chunks associated with the node. The number of bits required to record the length field is determined by the frequency F_{max} of the most common word on that node, hence $\lceil \log_2(F_{max}) \rceil$. The number of bits required to represent the offset is determined by the largest number of entries in an index and may be estimated as $(r_{chunk} \times N_w)/(N_n \times N_p)$. Total storage is thus estimated as

$$S_{SD} = \sum_{n=1}^{N_n} |V_n| \times (L_{max} \times B_{char} + N_p \times \lceil \log_2(\frac{F_{max} \times N_w \times r_{chunk}}{N_n \times N_p}) \rceil) \qquad (1)$$

Figure 2 shows the relationship of these structures diagrammatically.

4.1 Analysis of Query Processing

There are three basic algorithms in query processing: term location; proximity; and document ranking.

Term Location Algorithm for each Node

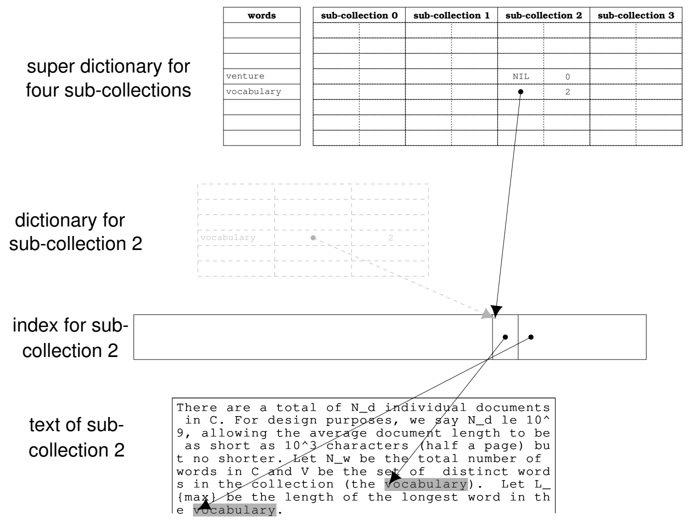
Step 1 Process the incoming search command (broadcast to all nodes).

Step 2 Locate the term in the sorted, memory-resident word list using a binary search.

Step 3 If there is an entry for the term in the word list, read the corresponding row from the superdictionary file. (One I/O operation per node.)

Step 4 For each of the sub-collections which contain occurrences, read a block of pointers into memory from the appropriate (pre-opened) index file. (Up to N_p I/O operations per node.)

Step 5 For each pointer transferred in step 3, write a tag indicating from which sub-collection it originated and update the relevance score of the document in which it occurs. The latter is performed during a synchronised scan of the occurrence pointers and the list of documents.



```
There are a total of N_d individual documents
 in C. For design purposes, we say N_d le 10^
9, allowing the average document length to be
 as short as 10^3 characters (half a page) bu
t no shorter. Let N_w be the total number of
words in C and V be the set of  distinct word
s in the collection (the vocabulary).  Let L_
{max} be the length of the longest word in th
e vocabulary.
```

Figure 2: Index, dictionary and superdictionary structures associated with a single processing node. The raw text of sub-collection 2 is represented at the bottom. It has been indexed and a dictionary has been prepared. The information from the latter has been merged into the super dictionary shown at the top. The dictionary may now be discarded. In this example, “venture” which occurs in at least one of the sub-collections but not in sub-collection 2, has no entry in the dictionary for sub-collection 2 but has a zero entry in the column of the superdictionary corresponding to sub-collection 2.

Step 6 Form a global count of the number of occurrences and return it to the front-end.

Step 1 involves communication from the user interface and the time to complete it is essentially constant ($t_1 = c_1$). Step 2 requires no I/O and no communication. The time to complete it depends upon the log of the amount of data per node. ($t_2 = m_2 \times \log_2(S_C/N_n)$)

Steps 3 and 4, if performed, are I/O intensive and require between two (one super dictionary and one index read) and $(N_p + 1) \times N_n$ (one super dictionary and N_p index reads per node) I/O operations. The data which must be transferred comprises N_o index entries, taking a time in seconds estimated as:

$$\frac{N_o \times \lceil \log_2(r_{chunk} \times S_{chunk}) \rceil}{BW}$$

Total I/O time is therefore expected to be:

$$t_{3\&4} = \frac{(N_p + 1) \times N_n}{I} + \frac{N_o \times \lceil \log_2(r_{chunk} \times S_{chunk}) \rceil}{BW} \tag{2}$$

Step 5 involves no I/O. The amount of CPU time required to perform it should be linear with the number of occurrences ($t_5 = m_5 N_o$). Step 6 requires a single global summation and a communication to the user interface ($t_6 = t_{gsum} + c_6$).

Proximity Algorithm for each Node

A number of sorted, memory-resident lists of pointers are sequentially scanned to locate values lying within a range of each other. The time taken is proportional to the sum of the lengths of the lists.

Document Ranking Algorithm

Step dr1 Each node sorts a complete list of document identifiers in order of decreasing accumulated relevance score. Then, if the N_r most relevant documents were requested, steps dr2-dr4 are repeated N_r times.

Step dr2 A global maximum is computed of the highest scores from each node.

Step dr3 The first part of the document is read from disk and scanned for the document identifier.

Step dr4 The document identifier and the score are communicated to the user interface.

The time for step dr4 is a small constant and may be neglected. The CPU component of the time taken is made up of the times for step dr1 and for the scanning part of dr3. The time taken to scan the head of the document may be approximated as a constant, c_{dr3}

$$t_{dr.cpu} = m_{dr} \times (N_d/N_n) \times \log_2(N_d/N_n) + N_r \times c_{dr3}$$

Step dr3 of the algorithm does not exploit I/O parallelism as only one document is selected and read at a time. Consequently the I/O time is dominated by the necessity to perform N_r seeks, each taking t_{seek} seconds. The I/O time including the communication is thus approximated as:

$$t_{dr.io} = N_r \times (t_{gmax} + t_{seek})$$

4.2 Analysis of Data Structure Building

The index construction algorithm used in the experimental work reported below keeps both raw data and index in primary memory during construction. A linear scan of the data in a chunk locates all wordstarts (transitions from non-alphabetic to alphabetic characters) and makes an array of pointers to them. These pointers are sorted such that the case-folded words they point to are in ascending order. A single scan through the index array then suffices to create the dictionary for the chunk. The CPU cost of index building was empirically studied in [5] and found to lie between linear and $n \log n$ approximations. The I/O costs are those of reading in the raw data and writing out the data structures at the end.

Super dictionary construction is essentially a series of pairwise merges of the dictionaries for each sub-collection. Both the CPU cost of each merge and the amount of data read and written should be directly proportional to the sum of the sizes of the partial super-dictionary and the new dictionary. As there are N_p merges (the first a trivial copy), the total data read is given by:

$$\frac{S_{SD}}{N_p} \times \sum_{i=1}^{N_p} i$$

where S_{SD} is given in equation 1 above. An identical amount of data is also written. Computational time should be directly proportional to the amount of data.

5 Experiments With A Prototype Implementation

A recent version of PADRE (the Parallel Document Retrieval Engine) [1] for the Fujitsu AP1000 implements a prototype of the model described above. Compression has not been implemented and considerable scope exists for space and possibly time improvements.

Table 1: Values of parameters fixed by the hardware configuration or by implementation decisions.

Parameter	Symbol	Value in the Experiment
Number of Nodes	N_n	128
Character Representation	B_R	8
Representation of Relevance Scores	B_R	32
No. of Sub-collections	N_p	6
Maximum Word Length	L_{max}	24

Table 2: Parameter values empirically determined for the prototype implementation using the test collection and the test configuration.

Parameter	Symbol	Value
Collection Size	S_C	3.33×10^9 char.s
Number of Documents	N_d	1.1×10^6
Number of Words	N_w	5.0×10^8
Collection Vocabulary Size	$\|V\|$	8.7×10^5
Average Chunk Size	$S_C/N_n/N_p$	4.34×10^6
Chunk Imbalance	r_{chunk}	1.31
Speed of Data Structure Building	v_{dsb}	2.1×10^6
Aggregate Disk Bandwidth	BW	5×10^7 char.s/sec.
Total Disk I/O Rate	I	3×10^3 op.s/sec.
Time to Compute Global Sum	t_{gsum}	175×10^{-6} sec.
Time to Compute Global Maximum	t_{gmax}	233×10^{-6} sec.

Experiments have been conducted with a small collection (3 gigabytes) on an AP1000 configuration comprising 128 SPARC 1+ nodes, each with 16 Mbytes of local primary memory. Thirty-two of the nodes have 500 Mbyte disks connected via a SCSI interface. Due to competing demands on the disks, a total of only 10 gigabytes of space was available for the test.

CDs 1-3 of the TREC [4] collection, with manual indexing information removed, constituted the test collection. The test query is shown in figure 3. It requires the location of 3 groups of alternative literal patterns anchored at word starts (index points), the computation of a 3-way proximity relation (`near`) between the groups, the assessment of relevance of all documents based on matches resulting from these operations and the creation of a ranked list of the 20 most relevant documents. It uses a total of 20 literal terms.

Tables 1 and 2 show the values of the model parameters which applied during the test.

5.1 Query Processing Performance

Using the current implementation, processing the task query over a six-component super dictionary was observed to take 10.23 sec., of which locating occurrences of the 20 terms took 9.21 sec., the proximity relation took 0.10 sec. and ranking the documents took 0.73 sec. All times reported are elapsed times as would have been observed by a user.

It was observed that the time to perform an unsuccessful search (zero occurrences), that is the time to perform steps 1,2 and 6 of the term location algorithm in section 4.1, was no more than 0.01 sec.

```
{proximity 1000}
{wsmode start}
{casesensitive 0}
{weight 0}
anyof "economic |economical |profit |profitable |profits |dollars "
anyof "recycle |recycling |recycles |reprocess |reprocesses
|reprocessing |conversion |converting |converts "
anyof "glass |paper |plastic |aluminum |cardboard "
{weight 5}
near 3
top 20
```

Figure 3: The test query used in experiments with the prototype.

Figure 4 shows that infrequently occurring terms are located quickly but that once a certain frequency is reached, location time is fairly constant up to the most frequent term encountered in this experiment.

Even for the most commonly occurring term, the total data transferred from the index was only 6×10^5 characters, corresponding to a transfer time of only 0.012 sec. Consequently, the number of I/O requests is the key component in the time taken for steps 3 and 4 of the term location algorithm due to the magnitude of seek delays and rotational latencies. Following the analysis in section 4.1 above, the number of I/O requests will be $7 \times 128 = 896$ for frequently occurring terms.

On the basis of the value given above for maximum I/O rate I, it would be expected that 896 reads should take 0.3 sec. but the observed time was 0.55 sec, presumably due to contention and uneven load balance between disks. In this application, then, a more realistic value of I is $I = 896/0.55 = 1.6 \times 10^3$ op.s/sec.

Considering the computational cost of step 5, the time taken to locate each of the 14 terms with a frequency of more than 5,000 (those very likely to occur in all of the $6 \times 128 = 768$ chunks) ranged only from 0.49 - 0.59 sec. The bottom three of these observations averaged 6,500 occurrences and 0.56 sec. to locate whereas the top three averaged 110,000 occurrences and also 0.56 sec. to locate. It is concluded that the CPU time for this step was being masked by variations in I/O time and was negligible for the range of term frequencies encountered in the test.

5.2 A Counter-Intuitive Method for Accelerating Query Processing

Clearly, query processing in our test environment is I/O bound. Increasing I by a factor of four by providing each node with a disk could be expected to reduce time to process the test query to 3.48 sec.

Remarkably, the model suggests that a similar effect could be achieved by *reducing* the number of CPU nodes to 32! The explanation is as follows. The number of I/O operations is determined by the number of chunks ($N_n \times N_p$). This product may be reduced by decreasing either the number of nodes or the number of sub-collections. Reducing either will produce large gains in overall time taken, up to the point where the costs of the CPU intensive steps in the algorithm become comparable to the I/O costs. Note that reducing the number of I/O requests does not alter the total I/O data transferred. Consequently the time to actually transfer the data remains constant.

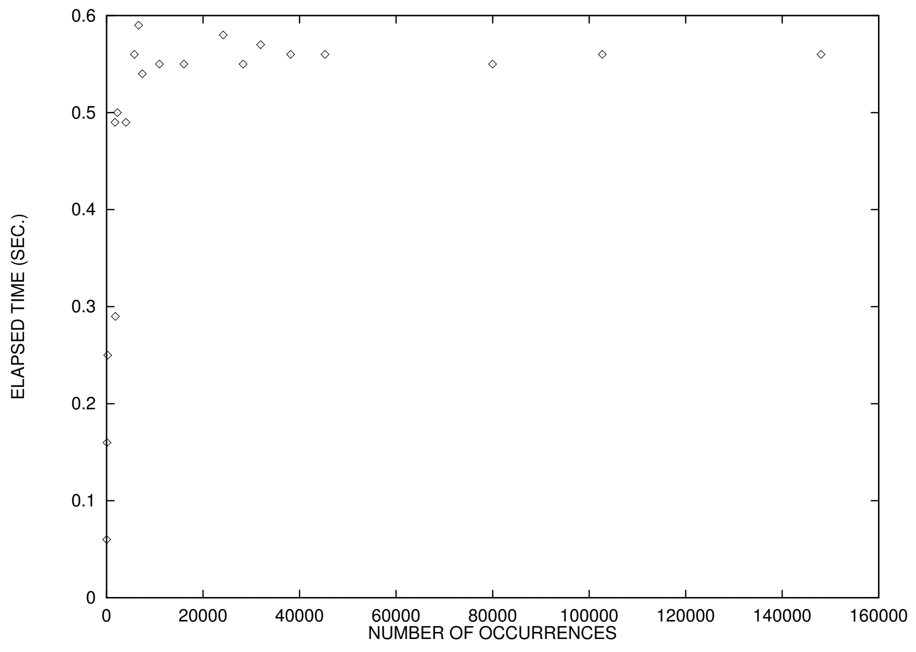


Figure 4: The relationship between term frequency and time to locate all occurrences for all the terms in the test query when processed over the test collection on the test machine.

5.3 Discussion of Parallelism in Query Processing

Had it been possible to index the test collection as 32 chunks (one per disk), at most one I/O request per disk would have been required to locate all occurrences of each term. In this case the I/O time to locate a term would of the order of 0.013 sec. instead of the 0.55 sec. observed, despite the fact that disk latency would still account for 75% of the time. Further increasing the degree of I/O parallelism would achieve limited benefits as query location time asymptotically approached disk latency. However, the ultimate goal of this work is to scale up to collections of the order of a terabyte in size and the amount of data to be transferred for a given query term is expected to scale with the size of the collection. Consequently, increasing the degree of I/O parallelism would be highly beneficial for a very large collection.

The amount of CPU time required to locate a query term is likely to be much smaller than the 0.01 seconds reported above as this figure was measured on the front-end and includes communications and time-sharing delays. Its contribution to the overall query processing time would not be expected to increase much above the present $20 \times 0.01 = 0.20$ sec. even if the number of processing nodes were reduced to match the degree of I/O parallelism.

However, the CPU intensive components of the proximity and document ranking algorithms would be expected to dominate when the degree of CPU parallelism had dropped to this level.

5.4 Data Structure Building

In the test a total of 2,127 sec. (elapsed time) was required to process the test collection. Of this, 1,800 sec. was required for index and dictionary building and 327 sec. for constructing the super dictionary.

It was observed that the breakdown between CPU time and I/O time in index and dictionary building was 1,560 to 240 respectively. Unfortunately, isolating the two components is more difficult in the case of super dictionary construction. The observed size of the super dictionary is 8.55×10^8 characters and, according to the analysis above, the quantity of data read and later written was $8.55 \times 10^8/6 \times 21 = 2.99 \times 10^9$ characters, giving a combined reading and writing transfer time of 120 sec. With the buffer sizes used, an estimated total of 32,500 separate I/O requests were issued, giving a time due to latency of 20 seconds (based on the revised I/O rate given earlier). The breakdown of CPU time to I/O time for super dictionary building is thus 187 to 140. Combining the two sets of CPU and I/O times gives an overall breakdown of 1,747 to 380. In summary, data structure building is heavily CPU bound, 82% of the total time being spent in computation.

6 Conclusions and Future Work

The performance of a parallel retrieval system is dependent upon many factors including speed and degree of parallelism of CPUs, memory and I/O system.

The algorithm presented above achieves speed in data structure building by dividing the collection into chunks small enough to be indexed entirely in primary memory. This strategy allows rapid response to collection changes, makes good use of parallelism and restricts the number of bits needed to represent pointers.

Unfortunately, in the experiment reported above, the strategy can be seen to inflict a heavy penalty during query processing due to the large number of I/O requests necessary to locate occurrences of terms in the query. It has been argued that this penalty may be reduced either by increasing the degree of I/O parallelism or by increasing the chunksize. In scaling up to a terabyte of data, both of these measures will be needed to ensure good performance and contain hardware cost.

Increasing the degree of I/O parallelism up to the number of chunks will give optimum query-processing performance, provided that there is adequate CPU speed. Reducing I/O parallelism below this level will cause I/O time to rise in inverse proportion to the number of disks. There is a trade-off between I/O speed in query processing and responsiveness to collection changes. The latter is essentially a function of the number of sub-collections.

The fully memory-resident nature of the current indexing algorithm is the root cause of the small chunksize and hence the large number of separate I/O requests in query processing. An algorithm which relaxed the constraint without significant penalty would be highly beneficial. One is currently under development. In the absence of such an algorithm it would be very productive to increase the amount of memory per node.

In the reported experiment, data structure building was seen to be CPU intensive. This is partly due to the fact that the nodes used were slow by current standards, being equivalent to Sun workstations dating from 1990. The optimum balance between CPU and I/O parallelism on a system handling a terabyte will obviously depend upon the power of the CPUs employed, the characteristics of the algorithms used and the relative costs of increasing CPU as opposed to I/O parallelism. If disk systems and nodes are provided in equal numbers, lack of CPU power is not likely to limit query processing performance.

Compression techniques deserve more consideration than they have been afforded here. On the basis of results elsewhere in the literature, compression may reduce by a factor of four the total disk space requirements.

Already, the model presented above has given several worthwhile insights into the running of the prototype implementation and how to improve it. More work is needed to refine and validate it further. Much larger experiments are necessary to confirm that the model and the algorithms will successfully scale to the terabyte level. To facilitate this, a project is under way to develop a significantly larger test collection.

Acknowledgments

The disk-based indexing and super dictionary machinery used in this paper were implemented by my colleague Peter Bailey to whom I owe considerable gratitude. The work also relies heavily on the HiDIOS parallel file system developed by Andrew Tridgell and David Walsh.

References

[1] Peter Bailey and David Hawking. A parallel architecture for query processing over a terabyte of text. Technical Report TR-CS-96-04, Department of Computer Science, The Australian National University, Canberra, `http://cs.anu.edu.au/techreports/1996/`, 1996.

[2] Janey Cringean, Roger England, Gordon Manson, and Peter Willett. Network design for the implementation of text searching using a multicomputer. *Information Processing and Management*, 27(4):265–283, 1991.

[3] Christos Faloutsos. Access methods for text. *ACM Computing Surveys*, 17(1):49–74, 1985.

[4] D. K. Harman, editor. *Proceedings of TREC-4*, Gaithersburg MD, November 1995. NIST special publication 500-236.

[5] David Hawking. The design and implementation of a parallel document retrieval engine. Technical Report TR-CS-95-08, Department of Computer Science, The Australian National University, Canberra, `http://cs.anu.edu.au/techreports/1995/index.html`, 1995.

[6] David Hawking and Peter Bailey. Parallel document retrieval engine (PADRE) web page. `http://cap.anu.edu.au/cap/projects/text\_retrieval/`, 1997.

[7] Mohammad Lofti-Jam and Alan Kent. Ranking on parallel computers using a compressed in-memory index. In Justin Zobel, editor, *Proceedings of the Australian Document Computing Symposium 1996*, pages 55–62, Melbourne, Australia, March 1996. Department of Computer Science, RMIT, Melbourne.

[8] S. F. Reddaway. High speed text retrieval from large databases on a massively parallel processor. *Information Processing and Management*, 27(4):311–316, 1991.

[9] Gerard Salton and Chris Buckley. Parallel text search methods. *Communications of the ACM*, 31(2):202–215, 1988.

[10] D.B. Skillicorn. A generalisation of indexing for parallel document search. Department of Computing and Information Science, Queen's University, Kingston, Canada (`skill@qucis.queensu.ca`), 1994.

[11] Craig Stanfill and Robert Thau. Information retrieval on the Connection Machine: 1 to 8192 gigabytes. *Information Processing and Management*, 27(4):285–310, 1991.

[12] Craig Stanfill, Robert Thau, and David Waltz. A parallel indexed algorithm for information retrieval. In *Proceedings of the ACM SIGIR Conference, Cambridge MA*. ACM, New York, 1989.

[13] Harold Stone. Parallel querying of large databases: A case study. *IEEE Computer*, 20(10):11–21, 1987.