

Parallel Search using Partitioned Inverted Files

*A. MacFarlane, *J.A.McCann, *⁺S.E.Robertson

*School of Informatics, City University, London EC1V OHB

⁺Microsoft Research Ltd, Cambridge CB2 3NH

{andym,jam,ser}@soi.city.ac.uk

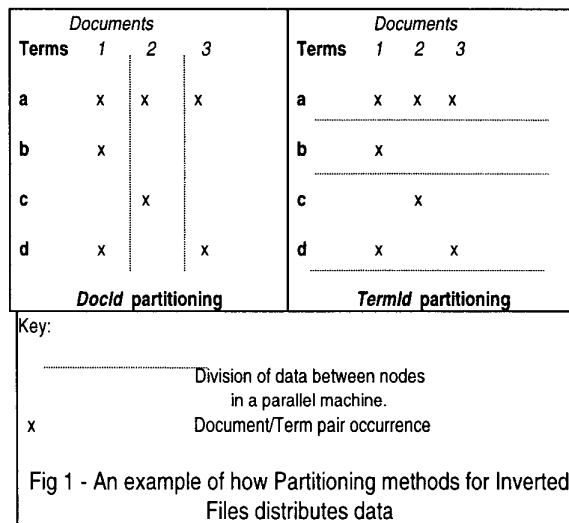
Abstract

We examine the search of partitioned inverted files with particular emphasis on issues that arise from different types of partitioning methods. Two types of index partitions are investigated: namely *TermId* and *DocId*. We describe the search operations implemented in order to support parallelism in probabilistic search. We also describe higher level features such as search topologies in parallel search methods. The results from runs on the two types of partitioning are compared and contrasted. We conclude that within our framework the *DocId* method is the best.

1. Introduction

In this paper we describe probabilistic search using PLIERS, a parallel information retrieval (IR) system based heavily on ideas developed on the Okapi system at City University [1]. The overall aim of the project is to discover which data distribution method is best for parallel IR systems. Our aim here is to compare search on inverted files given two types of partitioning methods: term identifier (*TermId*) partitioning and document identifier (*DocId*) partitioning. *TermId* partitioning is a type of partitioning which distributes unique word data to a single partition, while *DocId* partitioning distributes unique document data to a single partition. A partition is a physical division of the inverted file. A fuller discussion of these partitioning methods can be found in [2,3] and an example of these partitioning methods can be found in figure 1. The experimental aims and objectives of our research are given in section 2. The issue of search topologies is discussed in section 3, while section 4 gives a technical description of how the probabilistic search is supported on different partitioning methods. The hardware and software used in the experiments is outlined in section 5. The data and settings used for the experiments are described in section 6. The results of searches on the two chosen partitioning methods are reported in sections 7 and 8. Section 9 compares and contrasts these results with

each other and our stated aims/objectives. A summary and conclusion is given in section 10.



2. Experimental aims and objectives

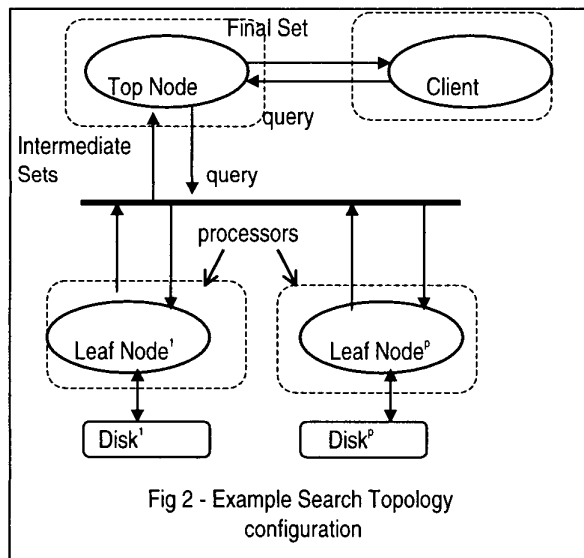
Earlier work on the subject of search performance on partitioned inverted files [2,4] used simulations. We use real Web collections and apply TREC queries to examine performance. The query model used in these simulations either assumes an equal probability of words occurring in a query [4] or uses both term skew and uniform term distribution models [2]. We believe that what the user does has a significant impact on the performance of a parallel system using the partitioning methods discussed. We put forward a hypothesis that focuses on query size: since users tend to submit smaller queries [5], only one or two of the nodes in the system will be servicing a query using *TermId*. Therefore for a given query the load balance will always be skewed as many nodes will be doing no work at all: this skew may accumulate with increasing numbers of queries. We define load balance as the spread of a given computation across a number of nodes in a parallel computer: ideally all nodes should have the same

computational load. In the *DocId* method however, each node is involved in the processing of all terms and therefore load balance for a single query is likely to be much superior to *TermId*. Our hypothesis is therefore;

The *DocId* partitioning method will perform better on probabilistic search because 1) users tend to submit short queries, which points to a load balancing problem for the *TermId* method, and 2) if documents are evenly distributed, *DocId* will produce good load balance as all nodes in the parallel machine process the query.

Our aim is to test the claims in the hypothesis as they relate to sequential query service. We do not address the issue of dynamically migrating indexes or concurrent query service. It should be noted that we principally address the issue of retrieval efficiency in this study, but mention retrieval effectiveness in order to verify that searching *TermId* and *DocId* partitioned indexes yield the same results.

3. Search topologies



In order to facilitate parallel searches on inverted files with differing data partitioning methods, we have implemented a generic search system which may be configured to use either partitioning method, consisting of a top node and one or more leaf nodes. The client process should be unaware of the distribution of data and retrieval responses with respect to effectiveness measures are identical on the different data partitioning methods. We

also have a requirement that the mapping of logical to physical topologies should be as flexible as possible. These components are described in below, followed by a discussion on search topologies. An example of how the components are combined can be found in figure 2.

3.1 Top Node

The main task of the top node in a search topology is to act as the interface for a client to the topology. It accepts a query from the client, distributes it to all of its child nodes and awaits the results. Depending on the type of operation, it may sort the results ready for presentation to the client or merge partially ordered results from its child nodes.

3.2 Leaf Node

The leaf node looks after one partition of the inverted file. It keeps an in-core record of keywords that is searched when a query is received. The inverted lists are then built for each element of the query and merged together to form a final result set for this node. This result set (or sets) is sent to the top node. The number of required inverted file partitions defines the number of leaf nodes.

3.3 Discussion on search topologies

The example in figure 2 is a master/slave topology with a top node and n leaf nodes (each with its own disk). The query referred to in figure 2 is a set of keywords, while the sets are retrieved inverted lists. The service of a query is done as follows: the top node receives a query from a client and distributes it to leaf nodes 1 to n . The result set for that query is sent back in the inverse direction to the top node, merging as necessary. In the example the top node is mapped to a separate processor from the leaf nodes and the client is on another processor. We can map the top and client nodes to any of the available processors as we wish. For performance reasons we only allow one leaf node per processor. Mapping either the top node or client node either separately or together with a leaf node also has performance implications. We describe the algorithm used and its impact on a parallel program with a given partitioning method in the next section.

4. Probabilistic search on partitioned inverted files

The term weighting model supported in PLIERS is the Robertson/Sparck Jones probabilistic model [6,7]. There are four main tasks to search when using the probabilistic

model. Firstly we retrieve the document sets from disk and place them in core. Once we have the document sets we can then assign a weight to each word/document pair. The sets can then be merged to create a single result set for the required keywords. This set is then sorted in descending order ready for presentation to the user. We describe both the implementation and implication of parallelism on each of these phases below.

The retrieval of document sets from disk is a straightforward process and the operation is identical irrespective of partitioning method used. The effect however can be very different: retrievals for *TermId* partitioning require only one I/O request per term, while *DocId* requires *p* requests (where *p* is the number of partitions), but the set transfer time per term will be shorter with *DocId* because of parallelism and short postings lists. It has been pointed out by Jeong and Omeicinski [2] that there are performance trade-offs between I/O requests and data transfer time when retrieving sets from disk.

$CW(i,j) = \frac{CFW(i) * TF(i,j) * K1+1}{K1 * ((1-B)+(B*(NDL(j)))) + TF(i,j)}$ <p><u>Variables</u> <i>n</i> = The number of documents term <i>t(i)</i> occurs in. <i>N</i> = The number of documents in the collection. <i>CFW(i,j)</i> = Collection frequency weight: $\log(N) - \log(n)$. <i>TF(i,j)</i> = The number of occurrences of term <i>t(i)</i> in document <i>d(j)</i> [term frequency]. <i>DL(j)</i> = The total number of terms in document <i>d(j)</i>. <i>NDL(j)</i> = Normalised document length: $(DL(j) / \text{average } DL \text{ for all documents})$.</p> <p><u>Constants</u> <i>K1</i> = Modifies influence of term frequency. <i>B</i> = Modifies effect of document length.</p> <p>Equation 1 - The BM25 Term Weighting function</p>

In the calculation of weights method we have implemented a number of functions that have been applied by Okapi at TREC including *bm_0*, *bm_11*, *bm_15* [8] and *bm_25* [9] (*bm* means best match). All results reported in this paper were produced on the *bm_25* term weighting function (see equation 1). A very important aspect of term weighting is the issue of collection statistics with respect to partitioning methods. If we treat partitions of the inverted file as being part of a global database we need to exchange data between the partitions in order to

ensure that the collection statistics are consistent for every weighting. For example in order to calculate a collection frequency weight we need to add all occurrences of a term from all partitions when using *DocId* partitioning: such addition is not necessary when using *TermId* partitioning since the term frequency is available in one partition. We therefore need to adjust our term weighting operations to suit the partitioning method being used. It is possible to keep a global dictionary at the cost of extra space. Other statistics such as average document length and total number of documents in the collection are also affected. It should be noted that term weighting is possible on independent collections [9,10], but the discussion of this subject is outside the scope of our research: our aim is to address the issue of term statistics across partitioned inverted files and not the results merging problem.

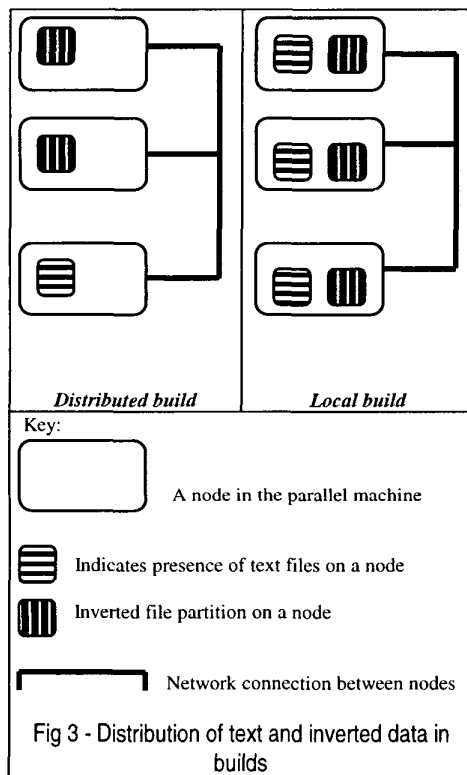
Once weights for each element of the set have been generated we can then use a PLUS set weight operation that is a special case of union: if two document identifiers are in both sets we add their weights otherwise we insert the unchanged posting record in the result set. Again different types of operations are required in differing partitioned methods. Using *DocId* partitioning we can merge local results, do a sort and send only the top *n* documents from that leaf to the parent node. With *TermId* partitioning we cannot merge the sets for the result until the top node has received data from all leaf nodes. This would appear to give *DocId* partitioning methods an advantage over *TermId* in that less communication is needed.

The final phase is simple in *DocId*: a simple multiway merge will produce the top *n* document required by the client. For *TermId* however we need to apply a sort to produce the top *n* documents. We can apply the sort either directly using a sequential sort or distribute the set elements amongst the leaves and apply parallelism to it: we have implemented both methods. Given that *DocId* can apply a sort in parallel in the third phase without any communication requirement, the method accrues a further advantage over *TermId*. Only the top *n* documents identified by the weighting operation are presented to the user.

5. Hardware and software used

The ParaLLeL Information rEtRieval Research System (PLIERS) has been developed at City University using ideas from Okapi to investigate the use of parallelism in IR. PLIERS is designed to run on several parallel architectures and is currently implemented on those which use Sun Sparc, DEC Alpha and Intel processors. All results presented in this paper were obtained on 8 nodes of a 12 node AP3000 at the Australian National University, Canberra. We also did runs on an 8 node Alpha farm to

check the portability of the code. Each node in these parallel machines has its own local disk: the *shared nothing architecture* [3] is used by PLIERS. The Fujitsu AP3000 is a distributed memory parallel computer using Ultra 1 processors running Solaris 2.5.1. Each node has a speed of 167Mhz with 128 Mbytes of memory. The machine we used has 12 nodes, but only 8 are available on a partition. The torus network has a top bandwidth of 200 Mbytes/s per second. For the Alpha farm, each node is a series 600 266Mhz Digital Alpha workstation with 128 Mbytes of memory running the Digital UNIX 4.0b operating system. Two types of network interconnects were used: a 155 Mbytes/s ATM LAN with a Digital GIGASwitch and a 10 Mb/s Ethernet LAN.



6. Data and settings used

The data used in the experiments comprised the BASE1 and BASE10 sub-sets of the official 100 Gigabyte VLC2 collection [12]. The BASE1 collection is 1 Gigabyte in size, while BASE10 is approximately 10 Gigabytes in size. The index structure we use is a conventional inverted file with a keyword and postings file split [13]. Our postings file can contain position data: we applied search to indexes with and without position data. We use two types of builds for indexes: *distributed*

builds where text is kept centrally and *local builds* where text is physically distributed to nodes and indexed locally. Builds are methods of parallel indexing and should not be confused with partitioning methods. Examples of how indexes and text are distributed between nodes are shown in figure 3. For the *distributed build* method we use the BASE1 collection only, creating indexes on 1 to 7 processors using both types of partitioning method and initiating searches on all of those indexes. The BASE1 and BASE10 collections were used for the *local build* method, running queries on 8 nodes. We use the *DocId* partitioning method only with *local build* indexes. We use different process to processor mapping strategies for each type of build. In *distributed build* we map the client and top node to a single processor, separate from each of the leaf nodes. For *local build* we have to map the client and top node to one of the leaf nodes because of restrictions on the available processors of the AP3000 partition set.

Collection	BASE1	BASE10
p@5	0.244	0.324
p@10	0.178	0.282
p@15	0.149	0.273
p@20	0.130	0.264
K1	1.5	1.5
B	0.2	0.2

Table 1 - Retrieval Effectiveness Results for *title only* experiments

Collection	BASE1	BASE10
p@5	0.188	0.356
p@10	0.172	0.298
p@15	0.145	0.271
p@20	0.128	0.247
K1	1.5	1.4
B	0.4	0.7

Table 2 - Retrieval Effectiveness Results for *whole topic* experiments

The queries are based on topics 351 to 400 of the TREC-7 ad-hoc track: 50 queries in all [12]. The terms were extracted from TREC-7 topic descriptions using an Okapi query generator utility to produce the final queries. We used two types of queries: one based on *title only* and one based on the *whole topic*. The average number of terms per query is 2.46 for *title only* and 19.58 for *whole topic*. The *whole topic* query set has 51 queries, one extra being for VLC2 experiment initialisation [12]. Our experiments concentrated on *title only* queries as users have a tendency to issue smaller queries. Tables 1 and 2 show retrieval effectiveness results on these queries

together with the *bm_25* tuning constants used. Retrieval effectiveness results for a given query type were identical on all runs irrespective of inverted file type used, partitioning method applied or number of processors used.

Query Type	File Type	Time (Secs)
<i>title only</i>	NPOS	0.110
	POS	0.216
<i>whole topic</i>	NPOS	2.45
	POS	6.02

Table 3 - Uniprocessor Average Elapsed Times (BASE1)

Our timing methodology was as follows: we declare the average of 10 runs. We declare results from the AP3000, but did runs on the Alpha farm to check portability of the code using evaluations to check that precision was identical. Sequential run times for comparison with parallel runs are declared in table 3. Sequential runs were done on one node with a single search process. Note that runs with postings only data is signified by NPOS file type, while runs on indexes with position data are signified by the POS file type. Although we do not use position data in our weighting functions, we need to examine the effect of position data on probabilistic search performance as in a deployed system users may submit queries with adjacency operations or may request passage retrieval. We assume that the position information is stored contiguously with each posting.

For the *TermId* method three strategies for term allocation to partitions are used. The most basic of these is the WC method that allocates terms to partitions on the basis of word count in 100 buckets. Words are placed in buckets using a lookup derived from an English Dictionary. A heuristic is used to allocate terms as equally as possible on this basis: this heuristic works by calculating the average value for each of the 100 buckets and attempts to derive a distribution of buckets amongst nodes that is within a given criterion, currently with 10% of the average value: up to five iterations are used. The other two methods allocate on collection frequency (CF) and term frequency (TF) basis.

$$\text{Scalability} = \frac{\text{Average Query Response Time (Smaller Collection)}}{\text{Average Query Response Time (Larger Collection)}} \cdot \frac{\text{Data Size (Larger Collection)}}{\text{Data Size (Smaller Collection)}}$$

Equation 2 – Scalability Measurement

We use a number of metrics to measure the performance of our program: average query processing time in seconds, Speedup (time on 1 processors/time on n processors), efficiency (speedup/n processors), load imbalance LI (maximum processing time on node/average processing time on node), extra cost ratio in time of inverted files containing position data, overheads such as compute and wait time in the top node and scalability (see equation 2).

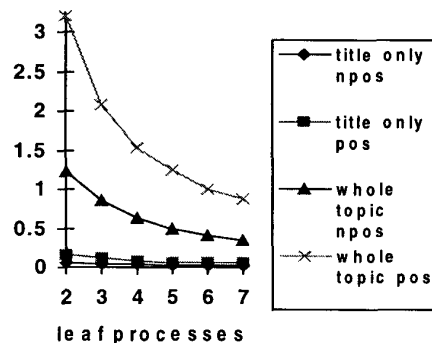


Fig 4 - Elapsed Time for *DocId* experiments

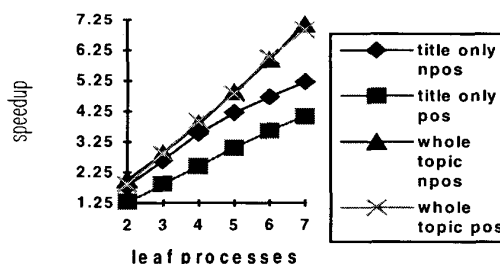


Fig 5 - Speedup for *DocId* Experiments

7. Search results from *DocId* partitioning

The results for this type of partitioning are encouraging. For *title only* queries the best elapsed times are very good indeed (see figure 4). Response times of 21 milliseconds are recorded for inversion with postings only data and 53 milliseconds for position data indexes. The comparison with VLC2 average response time is favourable: our *title only* time matches the best VLC2 results [12] at 2 processors and betters it on 3 to 7 processors. For *whole topic* the best elapsed times are a third of a second for postings only and 0.87 seconds for files with positions. All runs (including BASE10 runs) therefore meet the 10 second criterion for search times

suggested by Frakes [14]. Our BASE10 results are only bettered by one participant of VLC2 [12]. Time reduction relative to uniprocessor runs is exhibited by all multiprocessor runs.

The result is that good to reasonable speedup can be found on most parallel runs, being very near linear for *whole topic* and *title only* with postings only (see figure 5). Speedup on *title only* queries serviced on files with position data is disappointing however: the extra I/O needed to retrieve sets from disks outweighs any gain through parallelism on the sort for smaller queries. The efficiency results reflect the speedup figures and are particularly good for *whole topic* queries, results on all runs with this query type are very near 1 (see figure 6). Load imbalance is not an issue as for any query the load imbalance (LI) was found to be just over 1 in all cases (see figure 7). Imbalance was more noticeable on *whole topic* queries than *title only*, however.

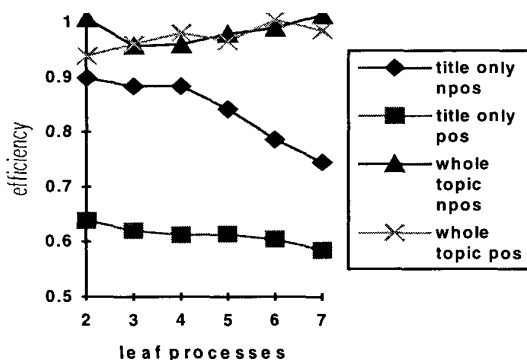


Fig 6 - Parallel Efficiency for *DocId* Experiments

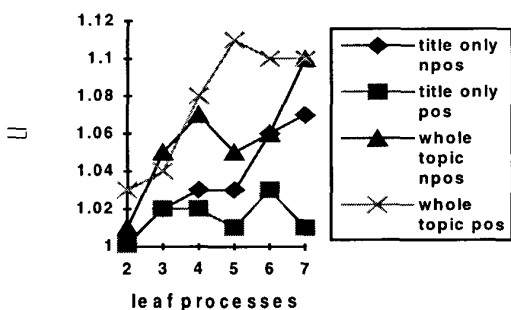


Fig 7 - Load Imbalance for *DocId* Experiments

Overheads found on the top process were found to be very small, in the single figure millisecond range for *title only* (see table 4). For *whole topic* they were insignificant with other aspects of the search dominating time: negative values are registered on some runs because of clock skew. The extra cost on search time for having position data in inversion was found to be constant for *whole topic*: search

was around 2.5 times longer than for postings only data. For *title only* the extra costs were worse on multiprocessor runs than uniprocessor runs and varied a great deal more than *whole topic*. This factor is further evidence of the extra burden I/O places on smaller queries with posting lists containing position data. The scalability of the parallel program on 8 processors is good with very good figures for *title only* queries (scalability is over 1) and for *whole topic* (see table 5).

Leaves	Title Only		Whole Topic	
	NPOS	POS	NPOS	POS
-				
2	1.25	1.84	-ve	-ve
3	1.93	1.54	-ve	324.3
4	1.17	2.43	-ve	1.03
5	2.88	2.99	10.69	-ve
6	3.44	2.83	-ve	10.97
7	3.56	4.29	6.83	10.26

Table 4 - Overheads in ms for *DocId* Experiments

Data	Title Only		Whole Topic	
	NPOS	POS	NPOS	POS
-				
BASE1 time (secs)	0.027	0.057	0.32	0.77
BASE10 time (secs)	0.18	0.54	4.18	6.45
Scalability	1.48	1.06	0.826	1.19

Table 5 - DOCID BASE1 to BASE10 Scalability Results on 8 Processors

8. Search results from TermId partitioning

The results with this type of partitioning are discouraging (results for runs on indexes with word count (WC) distribution are shown here: full details of other retrieval efficiency results can be found in appendices 1 and 2). While all the runs on any type of query meet the 10 second criteria for search times, there is little or no advantage in elapsed time for this type of partitioning method (see figure 8). Times do not compare favourably with VLC2 participant times [12]. The only example of an elapsed time decrease is *whole topic* queries serviced on indexes with position data, therefore slowdown is recorded in most cases (see figure 9). The effect on efficiency is dramatic with figures ranging from poor to unacceptable: a linear reduction in efficiency is recorded in all cases (see figure 10).

The implication of this is that scalability would be poor: we therefore did not do any experiments on the BASE10 collection. Load imbalance is generally very small for *title only* queries, but is perceptibly worse for

whole topic queries (see figure 11). It should be noted that these load imbalance (LI) figures are averaged over 50 queries on 10 runs so the evidence available suggests that computational skew does not accumulate with increasing numbers of queries. The overheads at the top process are clearly a problem with this type of method and are much higher than for *DocId* partitioning (see table 6). The overhead for *title only* queries run into the hundreds of milliseconds per query, while for *whole topic* they run to seconds. For costs of position data on probabilistic search it was found that the cost tend to reduce with increasing number of processors, a factor more marked in *title only* queries. The reason for this is that runs on indexes with position data gain more from I/O parallelism than runs on indexes with postings only.

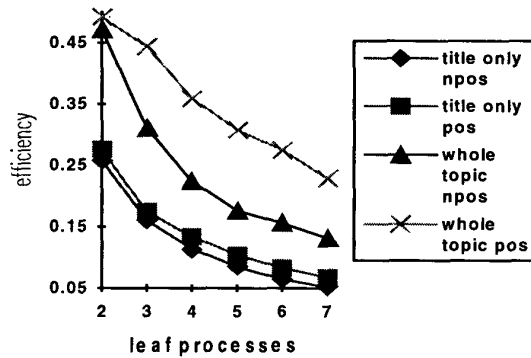


Fig 10 - Parallel Efficiency for *TermlD* Experiments (sequential sort: WC distribution)

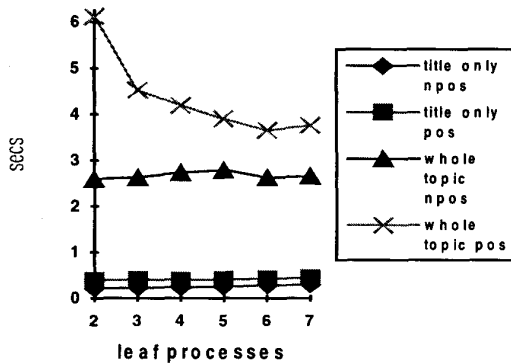


Fig 8 - Elapsed Time for *TermlD* experiments (sequential sort: WC distribution)

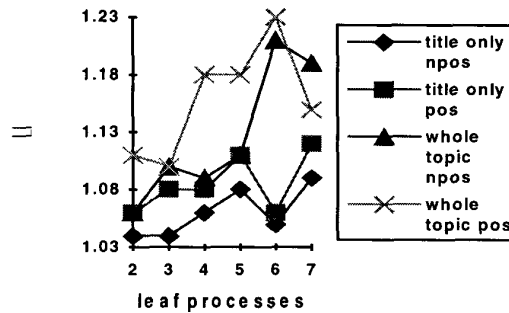


Fig 11 - Load Imbalance for *TermlD* Experiments (sequential sort: WC distribution)

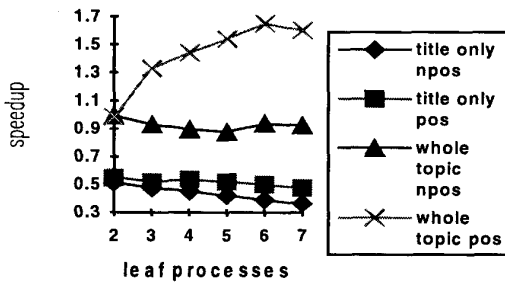


Fig 9 - Speedup for *TermlD* Experiments (sequential sort: WC distribution)

Leaves	Title Only		Whole Topic	
	NPOS	POS	NPOS	POS
-				
2	0.113	0.158	1.38	1.61
3	0.156	0.191	1.62	1.80
4	0.175	0.208	1.90	1.93
5	0.194	0.223	2.09	1.93
6	0.219	0.256	1.99	1.98
7	0.239	0.267	1.98	1.86

Table 6 - Overheads in secs for *TermlD* Experiments (sequential sort: WC distribution)

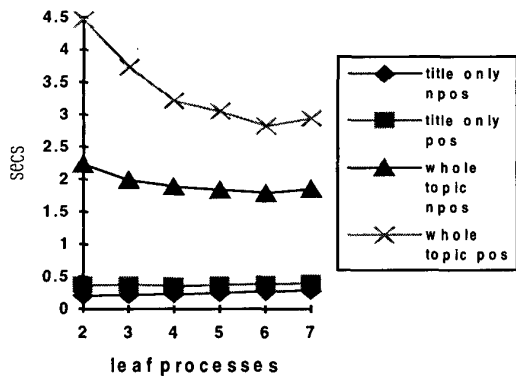


Fig 12 - Elapsed Time for *TermlD* experiments (parallel sort)

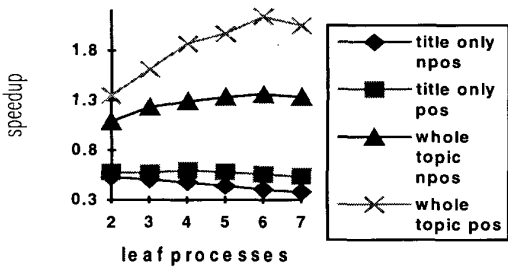


Fig 13 - Speedup for *TermlD* Experiments (parallel sort)

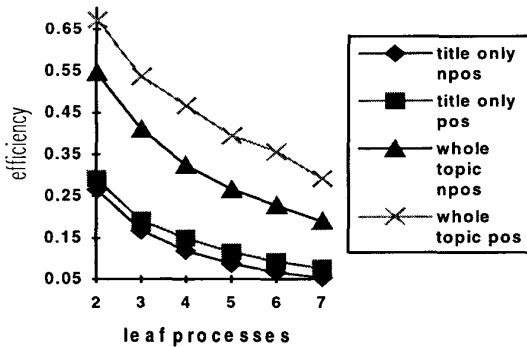


Fig 14 - Parallel Efficiency for *TermlD* Experiments (parallel sort)

Given the disappointing results described in the last paragraph, we thought it worth while having an attempt at speeding up the sort by using a parallel method. This required the generation of the final set as normal, but elements of the sort were scattered to leaves that applied a sort to their section of the final result set. The top ranked

20 documents were then gathered as per *DocId* method. The results however did not improve the performance of the parallel program much (see figures 12 to 15 and table 7). Average query processing times were reduced slightly, but speedup/efficiency for *title only* queries were still poor, while *whole topic* queries did show some slight speedup gains (efficiency was still poor). Load imbalance (LI) remained small on all runs. The basic problem with this revised method is that in applying the parallel sort, the amount of communication is increased such that most or all of the gain made in the parallel sort is lost.

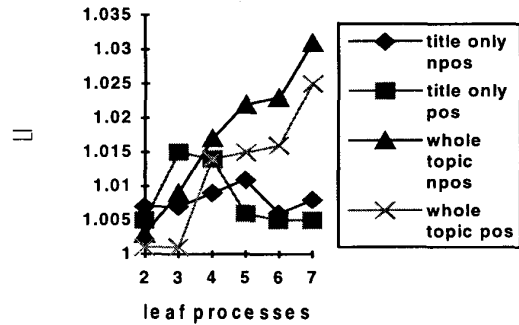


Fig 15 - Load Imbalance for *TermlD* Experiments (parallel sort)

Leaves	Title Only		Whole Topic	
	NPOS	POS	NPOS	POS
-				
2	0.307	0.111	0.200	-ve
3	0.552	0.381	1.51	0.820
4	0.646	0.428	2.15	0.377
5	0.684	0.655	2.84	0.828
6	0.826	0.734	4.03	1.44
7	0.813	0.783	3.28	1.06

Table 7 - Overheads in secs for *TermlD* Experiments (parallel sort)

9. Comparison of partitioning methods

In the context of the experiments defined in this paper, *DocId* partitioning is clearly the preferable method for probabilistic search using sequential query service. While both methods show acceptable response times, only *DocId* shows any real benefit when multiprocessors are used. *TermlD* partitioning clearly fails on parallel measures such as speedup and efficiency. The problem with *TermlD* is that too much data has to be communicated from the leaf to the top process and the sort cannot be parallelized without further communication between leaves and top

node. The reason for the communication overhead with *TermId* is that a final result for any given document cannot be guaranteed to be completed in a single leaf. The overhead at the top process is a serious bottleneck with *TermId* and cancels out any advantages in parallelism applied to I/O, set weighting merging and sorting. Unless some other topology could be found which would support the cluster of workstation model we use, we do not see *TermId* partitioning as a viable technique for parallelizing search for sequentially submitted queries. In order to support *TermId* we could perhaps use a more complicated topology. We do not see the point in attempting this when we have a partitioning method that works well on a simple topology. It could be argued that another type of architecture may be well suited to the *TermId* method. It is difficult to sustain such an argument given the dominance of the cluster of workstation model in parallelism today. The results presented in this paper demonstrate that *TermId* partitioning for distributed inverted files is not viable in our parallel search context.

How do the empirical results relate to the hypothesis stated in section 2; i.e. does the evidence refute or confirm the hypothesis? The second part of the hypothesis that suggests that good load balance searching on inverted files using the *DocId* partitioning method is confirmed. The first part on the *TermId* method needs to be revised however as it has been demonstrated that good load balance can be achieved as computational skew does not accumulate. The evidence applies to small queries only where the times are averaged over 50 queries. The imbalance for larger queries is much worse, but since users tend to submit smaller queries we cannot use that line of argument to defend the hypothesis as it stands. The good average load imbalance (LI) on *title only* queries are found irrespective of the term allocation to inverted file partition mechanism used. It is therefore possible that *TermId* may be able to offer a better concurrent query service than *DocId* particularly with respect to throughput. It should be stated however that although part of the hypothesis has been confirmed technological factors are more important than information theoretic ones in probabilistic search. The sort needed for ranking is a considerable cost, and *TermId* cannot apply parallelism without increasing communication costs, a disadvantage the *DocId* method does not suffer from.

A number of different assertions have been made with respect to partitioning search and we examine them given the evidence provided above. Jeong and Omiecinski [2] concluded that if the terms are less skewed in distribution *TermId* would be the best method to choose, while a skewed term distribution in queries would suggest that *DocId* would be a better partitioning method. Given that users tend to submit smaller queries it seems axiomatic that the *DocId* method would be preferable. However the

evidence produced above suggests other aspects of term weighting search are more important than the distribution of terms in the query. In particular the sort to produce the final ranked set: search time on *DocId* partitioning indexes is superior in this aspect. It should be stated that our results validate their assertion that searches on *DocId* partitioning indexes is better when term distribution is skewed. Tomasic and Garcia-Molina's [4] results indicate that a hybrid of the partitioning methods discussed in this paper yields better response times. While their results are not directly comparable with ours we can make assertions on a higher abstract level. They use a slightly different architectural model in which each node has two disks attached to a cluster node rather than one in our model. Their simulation suggests that in the best performing strategies the data for one document was kept on one node, our results validate theirs. Local sorts can be applied using their strategies and the same reduced communication load can be obtained.

10. Summary and conclusion

We propose a hypothesis on search performance on parallel IR systems which suggests that query size is a significant factor. We describe our topology for parallel search that consists of a top process (a system interface for the client) and leaf processes that look after inverted file partitions. We then describe probabilistic search and how such is supported on this topology. We use 8 processors of an AP3000 multiprocessor in order to gather experimental results on two web collections BASE1 and BASE10.

The *DocId* partitioning method shows advantages over all measures and is clearly a good scheme for applying parallelism to information retrieval if such is needed. The second part of our hypothesis is confirmed by the results on *DocId* partitioning. However technological factors in parallel search, namely the importance of sort in term weighting search, appears to have a more significant factor on performance than information theoretic factors. We have used this method to good effect on a much larger collection than the ones discussed in this article in our TREC-8 experiments [15]. The *TermId* scheme does not work in our framework and given the simplicity and widespread use of the cluster model we use, we do not see the scheme as being a viable one for probabilistic search using sequential query service. However the load balancing evidence average over 50 queries supports any argument that concurrent query service using *TermId* partitioning may be useful. While the first part of our hypothesis on search performance put forward in section 2 may be correct for a single query, our evidence suggests it could be false in concurrent query service. The direction for any further research in this area is to examine how the

inverted file partitioning methods affect the performance of parallel IR systems that offer concurrent query service in order to examine our hypothesis further. A more detailed analysis would be useful, examining such issues machine architecture, network qualities (contention, latency and bandwidth), ranking and qualities of user requests.

11. Acknowledgements

This work is supported by the British Academy under grant number IS96/4203. We are also grateful to ACSys for awarding the first author a visiting fellowship at the Australian National University in order to complete this research and use their equipment. We are particularly grateful to David Hawking for making the arrangements for the visit to the ANU.

References

- [1] S. E. Robertson, "Overview of the OKAPI projects", *Journal of Documentation*, 53 (1), January 1997, pp. 3-7.
- [2] B. Jeong, and E. Omiecinski, "Inverted file partitioning schemes in multiple disk systems", *IEEE Transactions on Parallel and Distributed Systems*, 6 (2), 1995, pp. 142-153.
- [3] A. MacFarlane, S.E.Robertson, and J.A.McCann, "Parallel Computing in Information Retrieval - An updated Review", *Journal of Documentation*, 53 (3), June 1997, pp. 274-315.
- [4] A. Tomasic, and H. Garcia-Molina, "Performance of Inverted Indices in Shared-Nothing Distributed Text document Information Retrieval Systems". *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, IEEE Computer Society Press, Los Alamitos, 1993, pp. 8-17.
- [5] C. Silverstein, M. Henzinger, H Marais and M. Moricz, "Analysis of a Very Large Web Search Engine Query Log", *SIGIR Forum*, 33 (1), Fall 1999, pp. 6-12.
- [6] S.E. Robertson and K. Sparck Jones, Simple, "Proven approaches to Text Retrieval", *Technical Report No. 356*, University of Cambridge Computer Laboratory, 1994.
- [7] S.E.Robertson, and K. Sparck Jones, "Relevance Weighting of Search Terms", *Journal of the American Society for Information Science*, May-June 1976, pp. 129-145.
- [8] S.E.Robertson, S.Walker and M.M. Hancock-Beaulieu, "Large test collection experiments on an operational, Interactive systems: Okapi at TREC", *Information Processing and Management*, 31, (3). 1995, pp. 345-360.
- [9] S.E. Robertson, S. Walker, S. Jones, M.M. Beaulieu M. Gatford and A. Payne, "Okapi at TREC-4", In: *D.K.Harman, ed, Proceedings of the Fourth Text Retrieval Conference, Gaithersburg, U.S.A, November 1995*, NIST, Gaithersburg, 1996, pp. 73-96,
- [10] A. Singhal, "AT&T at TREC-6", In: *E.M. Voorhees and D.K.Harman, eds, Proceedings of the Sixth Text Retrieval Conference, Gaithersburg, U.S.A, November 1997*, NIST, Gaithersburg, 1998, pp. 215-226.
- [11] G.V.Cormack, C.L.A.Clarke, C.R.Palmer, and S.S.L.To, "Passage-Based Refinement (MultiText Experiments for TREC-6)", In: *E.M. Voorhees and D.K.Harman, eds, Proceedings of the Sixth Text Retrieval Conference, Gaithersburg, U.S.A, November 1997*, NIST, Gaithersburg, pp. 303-320.
- [12] D. Hawking, N. Craswell and P. Thistlewaite, "Overview of TREC-7 Very Large Collection Track", In: *E.M. Voorhees and D.K.Harman, eds, Proceedings of the Seventh Text Retrieval Conference, Gaithersburg, U.S.A, November 1998*, Gaithersburg, NIST, Gaithersburg, pp. 257-268.
- [13] D. Harman, E. Fox, R.A. Baeza-Yates and W. Lee "Inverted Files", In: *W.B. Frakes, and R. Baeza-Yates, eds, Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, New Jersey, 1992, pp. 28-43.
- [14] W.B. Frakes, "Introduction to Information Storage and Retrieval Systems", In: *W.B. Frakes, and R. Baeza-Yates, eds, Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, New Jersey, 1992, pp. 1-12.
- [15] A. MacFarlane, S.E. Robertson, J.A. McCann, PLIERS at TREC8, In: *E. Voorhees, ed, Proceedings of the Eighth Text Retrieval Conference, Gaithersburg, U.S.A, November 1999*, NIST, Gaithersburg, 2000, (too appear).

APPENDIX 1 - TERMID Retrieval Efficiency Results - CF Distribution

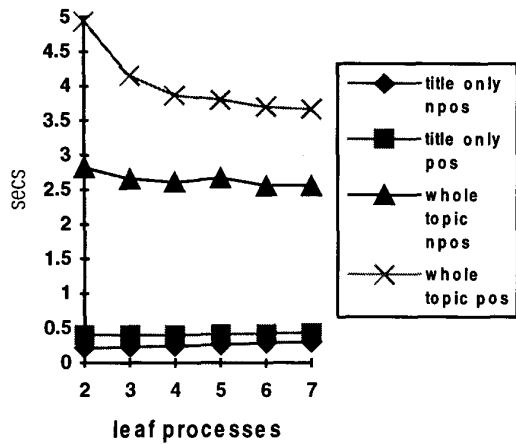


Fig 16 - Elapsed Time for *TermlD* experiments (sequential sort: CF distribution)

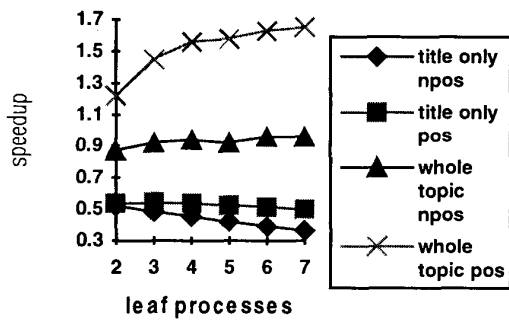


Fig 17 - Speedup for *TermlD* Experiments (sequential sort: CF distribution)

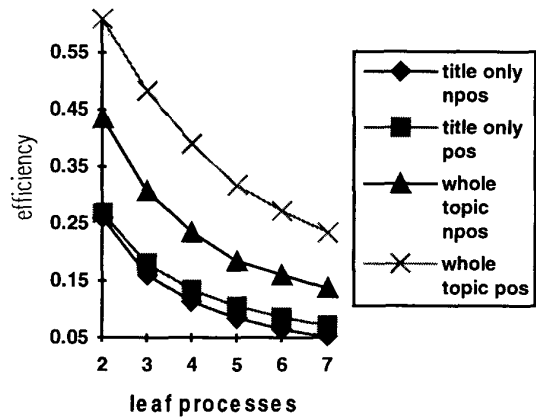


Fig 18 - Parallel Efficiency for *TermlD* Experiments (sequential sort: CF distribution)

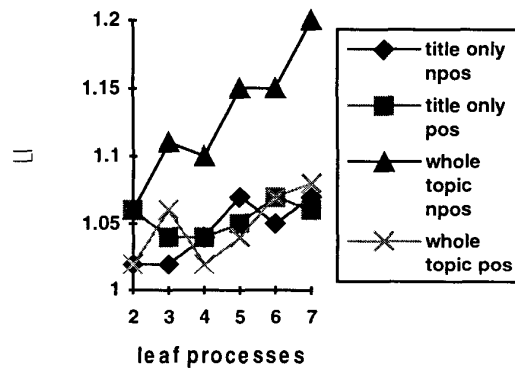


Fig 19 - Load Imbalance for *TermlD* Experiments (sequential sort: CF distribution)

Leaves	Title Only		WholeTopic	
	NPOS	POS	NPOS	POS
-	0.134	0.162	1.54	1.70
2	0.157	0.190	1.67	1.84
3	0.176	0.211	1.80	2.10
4	0.196	0.227	1.93	2.18
5	0.219	0.244	1.91	2.22
6	0.240	0.263	1.88	2.27

Table 8 - Overheads in secs for *TermlD* Experiments (sequential sort: CF distribution)

APPENDIX 2 - TERMID Retrieval Efficiency Results - TF Distribution

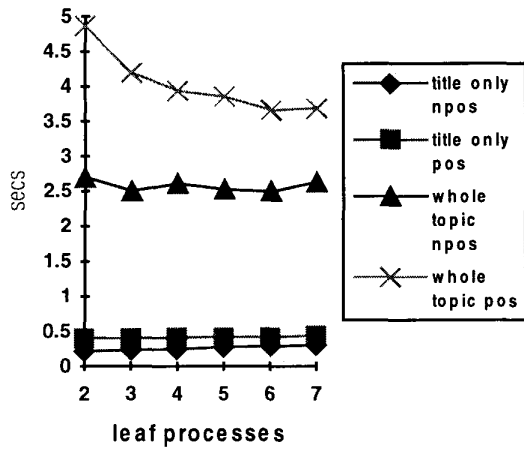


Fig 20 - Elapsed Time for *TermlD* experiments (sequential sort: TF distribution)

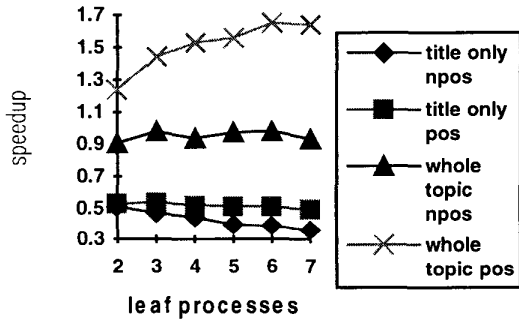


Fig 21 - Speedup for *TermlD* Experiments (sequential sort: TF distribution)

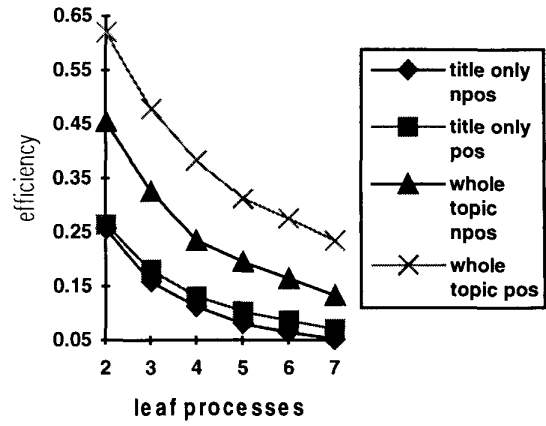


Fig 22 - Parallel Efficiency for *TermlD* Experiments (sequential sort: TF distribution)

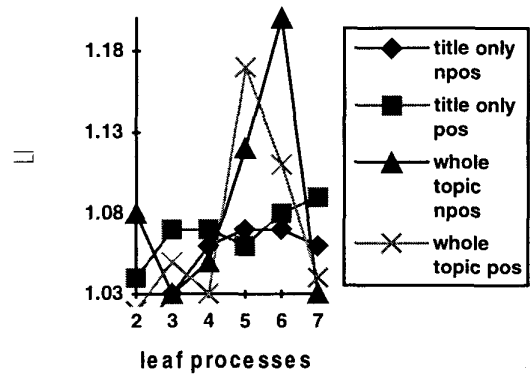


Fig 23 - Load Imbalance for *TermlD* Experiments (sequential sort: TF distribution)

Leaves	Title Only		WholeTopic	
	NPOS	POS	NPOS	POS
-				
2	0.138	0.168	1.42	1.65
3	0.157	0.189	1.67	1.86
4	0.177	0.216	1.92	2.16
5	0.197	0.234	1.85	2.09
6	0.218	0.239	1.89	2.12
7	0.241	0.260	2.13	2.34

Table 9 - Overheads in secs for *TermlD* Experiments (sequential sort: TF distribution)