A Distributed-Memory Algorithm for Lexicon Building

David Hawking
Co-operative Research Centre For Advanced Computational Systems
Department Of Computer Science
Australian National University
dave@cs.anu.edu.au

February 19, 2007

Suggested Running Head

A DISTRIBUTED MEMORY ALGORITHM FOR LEXICON BUILDING

Abstract

A parallel algorithm for preparing word frequency concordances over two specified sets of documents from a collection is presented. Good parallel efficiency is demonstrated on a 128-node distributed memory machine using sets whose combined size exceeds one gigabyte. It is demonstrated that efficiency is heavily influenced by hashing and communication strategies. A two-stage hashing algorithm is proposed to reduce communication overhead. Ways of increasing capacity are considered and the applicability of the algorithm to other text-processing functions such as index and symbol-table building is outlined.

FIGURE CAPTIONS

FIGURE 1:
The architecture of the Fujitsu AP1000. The 2-D torus network (T-net) links the processing nodes (cells). The Broadcast network (B-net) links all nodes and the front-end computer (host). Hardware speeds are 25 MBytes/sec. for each T-net link and 50 Mbytes/sec. for the B-net. There is also a dedicated status and synchronization network (S-net). Some nodes are directly connected to local SCSI disks via DDV (Distributed Disk and Video) option boards. The HiDIOS high-performance parallel file system [?] allows all nodes to access the disks at very similar speeds. AP1000 nodes run a single-user operating system called *cellos* though a full Unix (AP/Linux) is now available for later models of the AP1000 [?]. All communication with the external world is via the host.

FIGURE 2:
Node m has previously scanned the documents in set 1, sent local hash data to their respective global destinations and re-initialised its local hash table. At the point in time shown here, scanning of set 2 has been completed and the local hash table shows data extracted from the set 2 documents on this node. Two of the words (`koala` and `wombat`) from the local hash table hash to the same node (n) in the distributed global hash table. Their information is sent to node n in a single message, as shown. The segment of the global hash table held on node n contains the document counts for both set 1 and set 2 for words such as `koala` and `wombat`. The word `kangaroo` was found in at least one of the set 1 or set 2 documents on node n but its global repository is elsewhere. The word `rosella` was not found in the set 1 or set 2 documents on nodes m or n but hashes to the segment of the global table on node n.

FIGURE 3:
Storage organisation within a node.

FIGURE 4:
Elapsed time to form a null/full wordlist over a collection plotted against the size of the collection in words, using the RPR collision handling strategy. Also shown is an upper bound on the amount of the elapsed time which was consumed in communication.

FIGURE 5:
Elapsed time to form a null/full wordlist over a collection plotted against the size of the collection in words, using the LP collision handling strategy. Also shown is an upper bound on the amount of the elapsed time which was consumed in communication.

1 Introduction

The specific problem addressed in this paper is as follows:

> *Given two arbitrary sets of documents D_1 and D_2 distributed across a parallel distributed memory machine, form a list of all distinct words in the vocabulary $V_{1\cup 2}$ of $D_1 \cup D_2$. For each word w_i in $V_{1\cup 2}$, include counts f_1 and f_2 of the number of documents in each set which contain it. A word is any maximal sequence starting with a letter and containing only letters or digits.*

Word frequency correlation information finds application in document retrieval, in linguistic studies of word use and in testing authorship claims (or denials, as in a recent high-profile case).

More generally, a solution to the stated problem requires the construction of a lexicon of all the distinct words or symbols in a specified body of text and the association of application-specific information (in this case f_1 and f_2) with each of the entries. Consequently, the same basic algorithm may be used not only to solve the specific wordlist problem but also to form concordances of documents or collections, to build inverted-file indexes for search engines or even to build symbol tables from program text. For large data sizes, the computing resources required to solve these problems are significant. The present paper addresses the question of whether the large total memory size and processing power of distributed-memory parallel machines can be effectively brought to bear or whether the potential advantages are lost through communication overheads.

The specific problem stated above arose while investigating ways of augmenting text retrieval queries. It was hypothesised that words tending to occur much more frequently in documents known (or suspected) to be relevant to a research topic may constitute good discriminating features for finding further relevant documents. For example, the word `riboflavin` may occur 50 times as often in documents which match a query about vitamins as in randomly selected documents. Augmenting the query with `riboflavin` may enable retrieval of some additional documents, and usefully change the ranking of documents already identified as relevant.

Day [?] presents sequential programs for solving the word frequency problem based on linear search of the word table. He also discusses the use of binary search, binary trees, hash tables and tries to improve their efficiency. Knuth [?] describes and analyses the performance of these and other similar methods. Smith [?] compares various packages for forming concordances and gives some detail of the algorithms they use. Inverted file construction is part of most commercial document retrieval systems [?] and a wide range of indexing algorithms and structures have been employed. Harman and Candela [?] and Moffat and Bell [?] describe methods which are more space and time efficient than their predecessors.

The present paper is not concerned with further analyses of sequential algorithms but rather chooses a well-known algorithm and focuses on the problem of achieving good performance on a distributed memory parallel machine. This area has not been covered to any great extent in the literature. Stanfill and collaborators at Thinking Machines Corporation [?] developed inverted file methods for the Connection Machine CM2 but did not discuss in any detail the methods used for inverting the original text.

The key challenge for a parallel distributed memory solution to the stated problem is to achieve high parallel efficiency while producing global information (the overall wordlist and the frequency information aggregated over all processing nodes) from information derived locally from the nodes. A naive implementation would create massive communication and achieve very low efficiency on most parallel and distributed systems.

The implementation and results reported here are oriented toward the Fujitsu AP1000, a MIMD distributed memory system with SPARC 1+ nodes, dating from 1990. Details of this system are given in [?] and summarised in figure 1. The key communication parameters for the MPI system for the AP1000 [?] are as follows. Measured user-memory to user-memory bandwidth per link is 2.69 MBytes/sec and average latency between nodes is 172 μsec.

2 Chosen Algorithm

The output required from the algorithm is global frequency information, aggregated over all nodes. It was decided to accumulate this information using a global hash table distributed over all nodes. In this model all nodes can efficiently determine, without communication, which node holds a particular entry in the global table, or would hold it if it existed. The hash function $\mathcal{H}_g(w)$ returns a two component vector $[n, o]$ in which n identifies the node holding the relevant segment of the hash table and o gives the offset within that part.

If the distributed global hash table were the only data structure employed, a huge amount of unnecessary communication would occur; A message would be sent for each occurrence of every word on every node, except when the word hashed to the node on which it occurred. Use of a simple buffer would make little difference because long runs of words destined for the same node are likely to be uncommon.

A scheme in which each node allocated a separate multiple-entry buffer for each possible destination would be much more effective in reducing the number of messages transmitted at the expense of a significant amount of memory. An undesirable effect of such a scheme is that the amount of memory required grows linearly with the number of nodes.

Buffering reduces the number of distinct messages but does not alter the total number of items sent. It is possible to reduce the latter by aggregating local information prior to transmission. For example, the information that the word `stevedore` occurs 30 times on a node may be transmitted as a single item rather than 30 items. Local aggregation requires the use of an efficient local lookup algorithm which could be based on either a tree structure or a hash table.

A hash table was chosen in conjunction with a hashing algorithm designed to produce long runs when the local data is transmitted to its global destinations. An overview of the algorithm follows:

On each node, documents in a set are sequentially scanned for words. Each word encountered is looked up in a local hash table using a local hash function $\mathcal{H}_l(w)$. The local table records the word and a count of local documents which contain it.

When document scanning is complete for the part of a document set held by a node, it rehashes the entries using the global hash function $\mathcal{H}_g(w)$ and using the *node* part of the result as destination address for a message.

During the hash table scan, nodes must check at appropriate intervals for incoming messages and transfer their items to the segment of the global hash table. Items in the incoming message include the pre-calculated global hash index.

The function $\mathcal{H}_l(w)$ and $\mathcal{H}_g(w)$ are chosen so that entries which will hash to the same cell in the global table are likely to be stored contiguously in the local table. Then, while scanning the local table, runs may be detected and sent to the destination node in a single message, using only a single multiple-entry buffer.

Figure 2 illustrates the operation of local and global hash tables.

2.1 Details of Hashing Scheme

Many different hashing schemes are possible. The one chosen has the following characteristics.

- Buckets are not used.
- The number of possible entries in each hash table is always a power of two.
- The fact that the number of nodes in an AP1000 must be a power of two is relied upon. The global hash function produces a binary number in which the offset is in the least significant (rightmost) bits and the node number in the bits immediately to its left.
- The basic hash function makes use of up to the first 13 characters of a word. The result is initialised to zero, and then, for each of the significant characters, the result is multiplied by 17 and the character is added. The suitability of this algorithm, which is similar to one used in the TCL/Tk library (Ousterhout [?]), will be measured in the experimental results reported below.
- Local hash values are computed identically to global hash values (node and offset combined) but the result is right-shifted to produce the correct number of bits.
- If a collision occurs, a new hash value is computed which is forced to be odd. This value is used as a rehash step which is repeatedly added to the original hash address (modulo the table size)

until the required word or an empty slot is found. Because the table size is a power of two, all odd numbers are relatively prime to it, hence every slot in the table will be visited if necessary. This collision handling strategy will be called RPR (Relatively Prime Rehash). An alternative, in which the colliding item is stored in the next available free space was also tried for the local table only. It will be referred to as LP (Linear Probing).

- A collision counter is used to determine when the table has become unacceptably (more than 95%) full.
- A word globally hashed to a particular node n always remains there. Collisions will never cause it to move on to another node.

2.2 Storage Requirements

For convenience, each new word encountered on a node (either by scanning local documents or by receiving messages during global hashing) is stored in a memory area reserved for strings. Words are null-terminated and truncated at some arbitrary maximum length, usually 24 characters. Local words start at the top of the hash tables (see figure 3) and grow upward. Local word storage is cleared once first set scanning is complete. Non-local words start at the top of available memory and grow downward.

Each entry in the global hash table contains a pointer to the string, and a document count for each of the two sets. The number of entries required is determined by the expected maximum global vocabulary size $|V_{max}|$. A size of

$$2^{\lceil \log_2(|V_{max}|) \rceil + 1}$$

should suffice unless $\mathcal{H}_g(w)$ produces a very uneven distribution across the nodes. The number of global hash entries required on each node is of course $\frac{1}{N_n}$ of this.

Four bytes are allowed for global document counts and for pointers, a total of twelve bytes per entry, whether used or not. These allowances constitute a considerable degree of over-engineering. If the number of distinct words assigned to the segment of the global hash table stored on node n is U_n, then the total storage required by the global hash table is approximately:

$$\frac{12 \times 2^{\lceil \log_2(|V_{max}|) \rceil + 1}}{N_n} + U_n \times (|\bar{w}| + 1)$$

where $|\bar{w}|$ is the average word length.

Each entry in the local hash table contains a pointer to the string, a document count, and the index of the last document containing an instance of the word. The latter is necessary to ensure that only one occurrence of a word is counted per document. Again four bytes are allocated for the pointer but only two bytes are used for document index and count, imposing the restriction that no term may occur in more than 65,536 documents per node. (If space were very restricted a single bit could be used instead of the last-document index, to indicate whether the word had been seen in the current document. However, it would then be necessary to clear the bits for all words at the start of each document. The same method could be used to increase the limit on number of documents per node without increasing space requirements.)

The size of the local hash table on a node must be considerably larger than that node's segment of the global hash table. This is because the vocabulary size of a sub-collection $\frac{1}{N_n}$ of the size of the total collection is considerably greater than $\frac{1}{N_n}$ of the total vocabulary size. The ratio of global vocabulary size to the largest node vocabulary size will be called r_v. For a 128-node machine and a 2 gigabyte collection, $r_v \approx 10$ has been observed.

The space required for the local hash table on the node with the largest vocabulary size is thus:

$$\frac{8 \times 2^{\lceil \log_2(|V_{max}|) \rceil + 1} + |V_{max}| \times (|\bar{w}| + 1)}{r_v}$$

2.3 Transferring Data from Local to Global Hash Table

Once a node has extracted words from all the documents it holds from a set, it proceeds to scan its local hash table and transfer items to the correct segment of the global hash table. Some items are stored locally and runs destined for a remote node are buffered for sending.

Each node begins scanning at the block of entries which are expected, barring collisions, to remain locally. The effect of this is that all nodes start scanning at a different place in the table thereby reducing the probability that nodes will be bombarded with simultaneous messages from a large number of other nodes. Scanning wraps around when the end of the table is reached.

Each node processes all outstanding messages whenever a buffer is sent to another node and whenever an item is transferred locally.

Once a node has finished scanning its local hash table a `FINISHED` message is multicast to all nodes. The local to global hash transfer algorithm terminates when a node has received $N_n - 1$ `FINISHED` messages. Barrier synchronisation calls cannot be used because nodes which have finished scanning their local table must remain ready to process incoming messages until all other nodes have also finished.

3 Experiments With A Specific Implementation

3.1 Design Parameters

The above algorithm has been implemented on a 128 node AP1000 with 16 Mbytes of memory per node as part of the PADRE document retrieval system. [?, ?] PADRE provides the necessary facilities for loading and searching documents as well as for specifying document sets.

Parameters were set assuming a maximum global vocabulary size of nearly half a million words, leading to a global hash table of one million entries or 8,192 entries per node. Global hash values comprised 20 bits, 7 to specify the node number and 13 the offset.

A value of $r_v = 8$ was used in setting local hash table size. It is the power of two closest to, and on the conservative side of, the previously observed value of $r_v \approx 10$. Accordingly, the local hash table on each node has a total capacity of 65,536 entries and requires a minimum of approximately half a megabyte.

Message buffers sent when transferring document frequency data from a local hash table to a remote segment of the global hash table may contain data for up to 100 words. If more than 100 words need to be sent from one node to another, multiple buffers are sent.

3.2 Performance Results

Performance measurements were conducted using subsets of the TREC [?] document collection. In order to heavily stress the system, a wordlist was computed using the full set of documents as one operand and the null set as the other. In processing this rigorous test, every document is scanned once, every word is scanned once and the resulting wordlist contains every distinct word in the collection.

Figures 4 and 5 show the time taken to form the null/full wordlist over collections of different sizes using the RPR collision algorithm and the LP algorithm respectively. Also shown is an upper bound on the time consumed in communication. This time is the maximum of the times taken by nodes to twice scan their local hash tables and distribute the items to the global hash table. It has been corrected by subtracting twice the time taken to scan an empty local hash table. The time is an upper bound on communications costs because it includes the computational component of message buffer assembly, incoming message processing and global hashing.

Table I gives details of the collections used.

3.3 Memory Use

The memory sizes observed for the null/full run over collection 3 are given in table II.

3.3.1 Effectiveness of Chosen Hashing Strategies

The effectiveness of the chosen hashing function may be measured by

1. The degree to which items in the global hash table are uniformly spread across the nodes. In the case of the null/full test on collection 3, the ratio of maximum to average entries was 1.17.

2. The longest chain of collisions for the local hash table, taking into account how full the table is. In the case of the null/full test on collection 3, the fullest table was 72.5% full and the longest chain of collisions was 50 for the RPR collision strategy and 1,556 for the LP method.

3. The longest chain of collisions for the segments of the global hash table, taking into account how full the segments are. In the case of the null/full test on collection 3, the fullest table was 69.7% full and the longest chain of collisions was 57.

3.3.2 Effectiveness of Communication Optimisations

The two design decisions which were intended to reduce communication load were the use of dual hash tables and the structuring of hash tables to increase the likelihood of runs with consequent possibility of buffering.

The effect of dual hash tables may be estimated as follows. If only the global table had been employed, every word in the collection would have required global hashing. To a very close approximation, $\frac{N_n - 1}{N_n}$ of these hashes ($\frac{127}{128}$ in this case) would have resulted in an item to be communicated. For the null/full test on collection 3, 160,927,105 items would have been communicated compared with the observed figure of 5,450,505. In other words, the number of items to be communicated across the machine was reduced by a factor of 30 at the cost of less than a megabyte of memory per node and some additional processing time.

The effect of the hash table structuring is measured in terms of the ratio of items sent to messages sent. As can be seen from the observed ratios reported in table III, the effectiveness is dramatically dependent upon the collision handling strategy in the local hash table. The RPR strategy is very effective at reducing collisions but the LP strategy quite dramatically improves communications performance. The latter has more effect on the overall running time.

4 Discussion

4.1 Robustness of the Algorithm

Early implementations of the algorithm were subject to occasional crashes due to system message buffers overflowing on one or more nodes. Due to variations in message arrival patterns, repeated identical runs might sometimes succeed and sometimes fail. To eliminate this unpredictability, a simple acknowledge protocol was introduced in which node m will not send another message to node n until the previous one has been processed and acknowledged. It was observed that significant increases in run-time occurred as a result of this change. It may be possible to reduce this additional overhead by increasing the buffer size, using a more sophisticated protocol or by improving the method of checking for incoming messages, for example by using signals.

4.2 Gains Through Parallelism

In calculating the speed-up of the parallel algorithm, implicit reference is made to a serial algorithm running on a hypothetical SPARC 1+ node with sufficient random access memory to avoid all need to access disk. The speed-up under this definition is the product of the observed efficiency given by

$$\frac{total\_time - communication\_time}{total\_time}$$

and the number of nodes.

Efficiency and speed-up values derived from the timing data reported above are shown in table IV. The speed-up factors observed on the 128 node machine are considered quite satisfactory given the necessity for significant communication.

The speed-up figures given in the rightmost column are relative to the LP collision-handling method on the sequential machine. However, in the sequential case, there would be no reason to use the LP with its higher collision rate. Accordingly, a more realistic estimate of speed-up may be the ratio of RPR compute time per node times number of nodes to the total parallel run time using the LP method. These ratios are shown in parentheses in the rightmost column of the table. The speed-up measured in this way drops off in the collection 3 case because the increase number of collisions emphasises the superiority of RPR in handling them.

Speed-up measures quoted assume an even balance of work across the nodes. This assumption holds quite well here because there was in fact a very even balance of data across nodes and a reasonably even distribution of the global hash table. If this had not been the case, then the sequential algorithm time would have been over-estimated and so would the speed-up ratio.

4.3 Size Limits

The sizes of both hash tables were fortuitously well-chosen, each being the smallest powers of two sufficient to accommodate the needs of the largest possible document sets. Sufficient space remained in each to avoid excessive collisions. The observed value of r_v for collection 3 was in fact 13.1 (for the node with the largest vocabulary).

It has been demonstrated that the algorithm as implemented is capable of computing wordlists over sets totalling a gigabyte on a 128 node machine with a total of 2 gigabytes of RAM. The present implementation is in fact capable of handling a very much larger collection, using searches over disk-resident inverted file indexes to select the document sets. The combined document sets must still be small enough to fit in available RAM, whose size is somewhat diminished because memory is consumed in supporting indexed searches.

Small increases in the maximum treatable set sizes may be possible through reducing the amount of memory allocated for system message buffers, trimming the size of the PADRE executable, pruning array sizes and adopting more efficient bit-packings. More significant gains could be made if the documents were kept in compressed form and decompressed only at the time of scanning. These gains would be made at considerable expense in time due to the cost of decompression.

The most attractive option for increasing set capacity is that of buffering the data off disk, in which case the capacity limit is close to that of the disk system. If I/O were overlapped with scanning, the time penalty associated with this increase in capacity would not necessarily be very high provided that sufficient nodes were equipped with disks. In this case the hash table sizes would need to be increased in proportion to the relative increase in $|V_{max}|$. Using one novel word per thousand new words as a rule of thumb, a doubling of collection size from one gigabyte to two should increase $|V_{max}|$ by only 26%.

4.4 Effectiveness of Hashing and Communications Algorithms

The basic hashing function is seen to yield a very even spread across nodes and the RPR collision handling algorithm appears to be very effective in reducing the maximum length of collision runs.

The use of dual hash tables has been seen to cut the number of items transmitted by a large factor. The efficiency of the algorithm would have been dramatically reduced had this not been done. In contrast, in the 128 node collection 3 case, using the local hash table memory for per-node buffers would have resulted in the transmission of 30 times as many items and 59 times as many messages. For a machine with more nodes the dual-hash strategy compares even more favourably.

The effectiveness of the strategy to buffer multiple items into single messages was poor when using the RPR collision handling algorithm. Considering the null/full task over collection 1, nodes have an average of 76 entries destined for each other node but the average number of items sent per message is only 6.5. The explanation is that whenever a collision occurs, the probability that a particular table entry will be selected in the initial rehash is equal for all entries. Consequently, there is a $\frac{127}{128}$ chance that the rehash will not lie in the same block of the table. A relatively small number of collisions per node suffices to seriously break up the runs. That this explanation is correct can be seen in the dramatic run-length improvement seen when the simpler collision strategy (LP) is adopted. It may be sensible to introduce buckets in combination with RPR collision handling in order to achieve a low collision rate but allow longer communication runs.

5 Conclusions

An efficient and robust parallel algorithm has been designed for the stated problem and tested on a 128-node machine. It has been shown to achieve good parallel speed-up and to be able to process document sets whose total size exceeds half the total memory capacity of the machine. The dependence of good parallel efficiency on local hashing strategy and upon message buffering has been demonstrated. Allowing scanning of larger disk-resident document sets would constitute a useful future extension. It is also planned to investigate the use of the same basic algorithms and structures in building inverted file indexes.

ACKNOWLEDGEMENT

The support of the ACSys Co-operative Research Centre and the ANU-Fujitsu CAP project is gratefully acknowledged. Prof. Richard Brent's advice on the manuscript was appreciated as were suggestions made by the anonymous referees. Thanks are due to the U.S. National Institute of Standards and Technology and to various copyright holders for access to the TREC data collection.

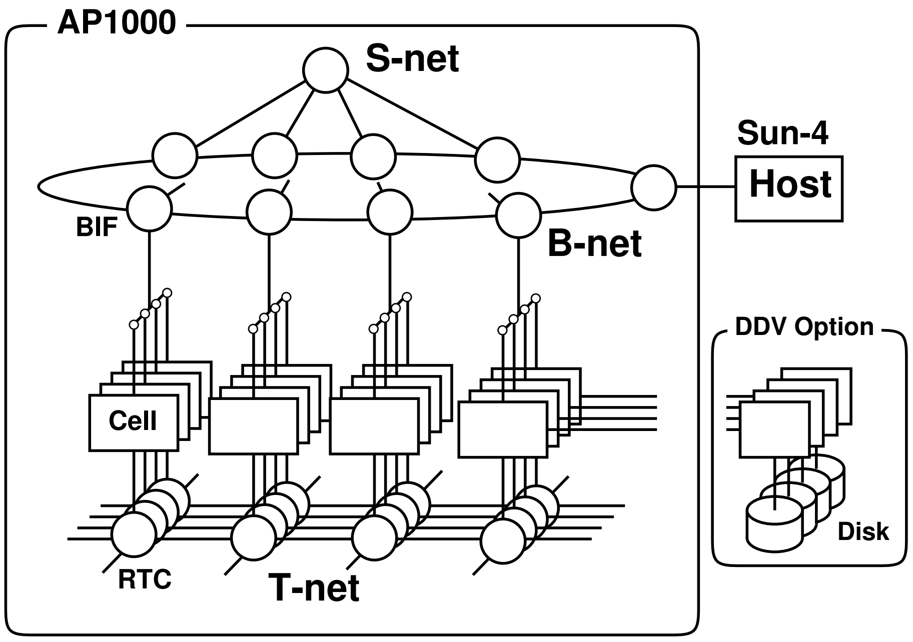

Figure 1:

Table I: Characteristics of collections used in reported experiments.

Coll.	No. Words	Distinct Wds	No. Documents	From
1	10,857,506	93,443	21,705	Wall Street Journal 1990
2	90,328,651	465,404	210,158	Financial Times
3	169,636,643	645,925	307,126	Financial Times, San Jose Mercury and US Patents

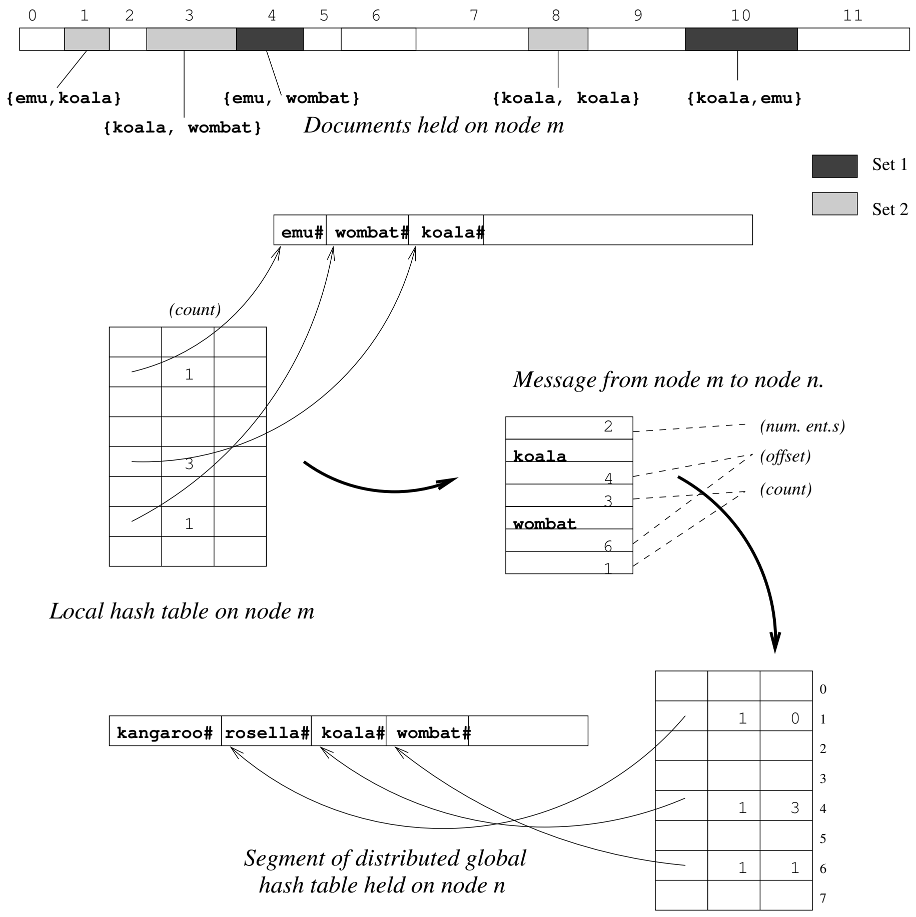

Figure 2:

Table II: Memory sizes observed during null/full run on collection 3 using 128 node Fujitsu AP1000 and RPR method. Sizes quoted refer to totals across the entire machine.

	Size (Mbyte)
Available space at runtime	1543
Collection size	1024
Space used in fixed tables	76
Space wasted in empty hash table entries	27
Free space on completion	254

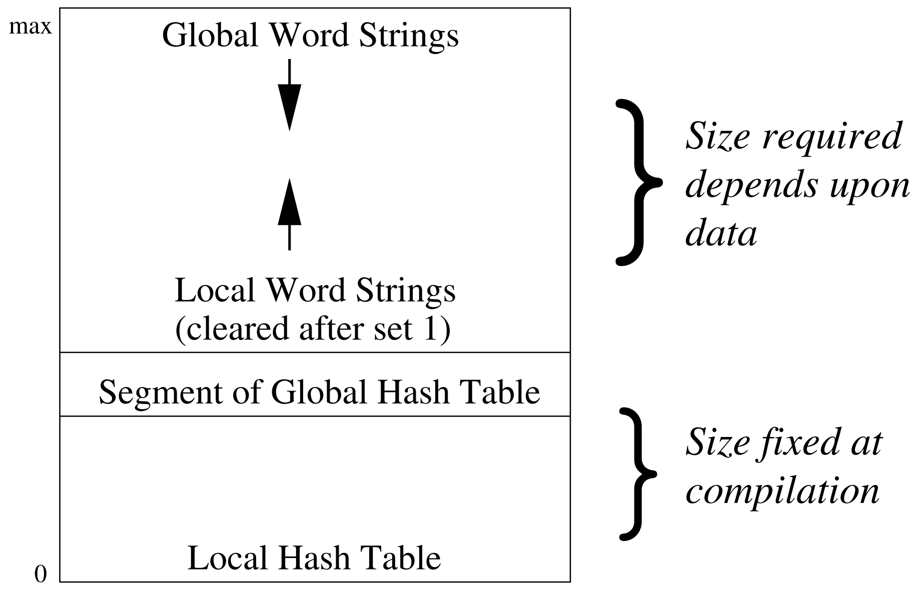
max
Global Word Strings
Local Word Strings
(cleared after set 1)
Segment of Global Hash Table
Local Hash Table
0
Size required depends upon data
Size fixed at compilation

Figure 3:

Table III: The trade-off between communications buffering effectiveness and collision avoidance for the null/full test over three different collections. Figures are shown for the two alternative collision strategies: Linear Probing (LP) and Relatively Prime Rehash (RPR)

Coll.	Items/Msg (RPR)	Max. Collis. (RPR)	Items/Msg (LP)	Max. Collis. (LP)
1	6.5	7	70.6	30
2	2.4	20	64.4	761
3	1.9	50	43.4	1,556

Table IV: Efficiency and speed-up data for the null/full task over different collections. Figures are given for the two alternative local hash collision strategies: Linear Probing (LP) and Relatively Prime Rehash (RPR). Note that the "communication" costs used here include all of the additional computational overhead occasioned by the dual hash tables (of which only one is necessary in the single-node case). Consequently, the figures given are considered to be reasonable estimates.

Coll.	Efficiency (RPR)	Speed-up (RPR)	Efficiency (LP)	Speed-up (LP)
1	0.46	58	0.68	87(88)
2	0.53	68	0.81	104(93)
3	0.54	70	0.80	102(76)

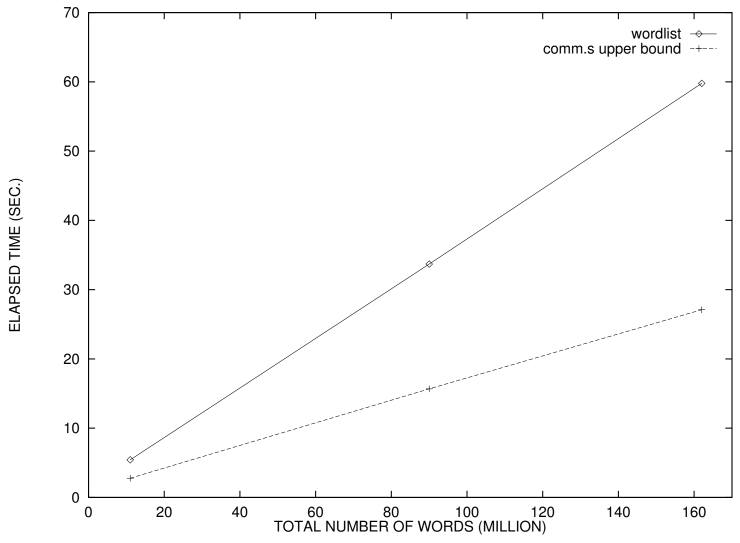

Figure 4:

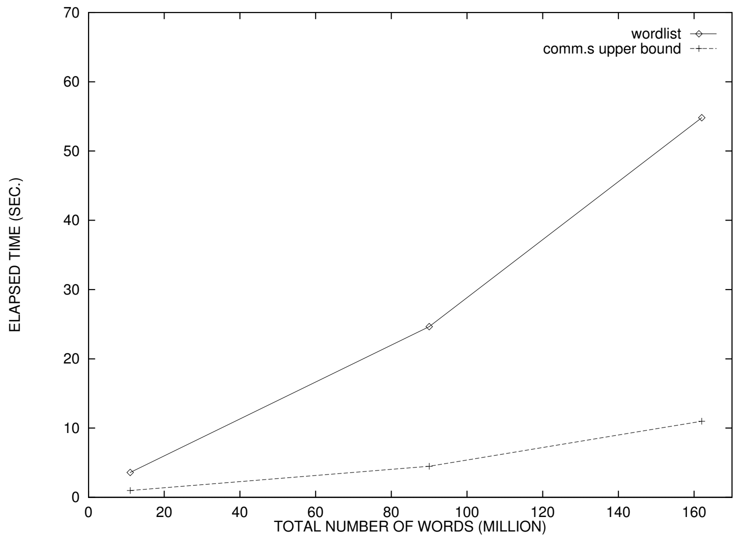

Figure 5: