THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-95-08

The Design and Implementation of a Parallel Document Retrieval Engine

David Hawking

December 1995

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

TR-CS-95-07 Raymond H. Chan and Michael K. Ng. *Conjugate gradient methods for Toeplitz systems.* September 1995.

TR-CS-95-06 Oscar Bosman and Heinz W. Schmidt. *Object test coverage using finite state machines.* September 1995.

TR-CS-95-05 Jeffrey X. Yu, Kian-Lee Tan, and Xun Qu. *On balancing workload in a highly mobile environment.* August 1995.

TR-CS-95-04 Department of Computer Science. *Annual report 1994.* August 1995.

TR-CS-95-03 Douglas R. Sweet and Richard P. Brent. *Error analysis of a partial pivoting method for structured matrices.* June 1995.

TR-CS-95-02 Jeffrey X. Yu and Kian-Lee Tan. *Scheduling issues in partitioned temporal join.* May 1995.

The Design And Implementation Of A Parallel Document Retrieval Engine

David Hawking
Co-operative Research Centre For Advanced Computational Systems,
Department Of Computer Science,
Australian National University, ACT 0200, Australia
phone: 61-6-249 3850, fax: 61-6-249 0010, e-mail: dave@cs.anu.edu.au

October 10, 1994[1]

[1]Date paper completed. Publication as a technical report was delayed for various reasons.

SUMMARY

Document retrieval as traditionally formulated is an inherently parallel task because the document collection can be divided into N sub-collections each of which may be searched independently. Document retrieval software can potentially exploit the power and capacity of a large-scale parallel machine to improve speed, to extend the size of the largest collection which can be processed, to respond quickly to changes in the document collection and/or to increase the power and expressivity of the retrieval query language. This paper includes discussion of the issues involved in the design of a practical parallel document retrieval engine for a distributed-memory multicomputer and a description of the implementation of PADRE, a retrieval engine for the Fujitsu AP1000. Performance results are presented and scope of applicability of the techniques is discussed.

KEYWORDS Parallel computing, Text, Document retrieval, Pattern matching

Introduction

Document Retrieval

Sifting relevant documents from among a large collection is of great importance in almost all professions, particularly investigative journalism, academic research, business, public administration and the law. Traditionally, such tasks have been carried out by human researchers or research assistants.

The increasing availability of computerised research collections, coupled with advances in computer technology, have given rise to the possibility of carrying out such research not only faster but better. Computerised retrieval systems are now able to perform complete searches over very large collections of text (provided it exists in electronic form), without reliance on manually generated indexing information.

Performance of a retrieval system is traditionally measured in terms of *precision* and *recall*, which are calculated by dividing the number of relevant documents retrieved by, respectively, the number of documents retrieved and the number of relevant documents actually in the collection. As reported by Harman[1], state-of-the-art retrieval systems now achieve quite acceptable performance on these measures.

Computerised document retrieval systems rank documents according to estimated relevance to the research topic in question. Precision is usually high at the top of the list but falls away as more documents are retrieved. By the time all relevant documents have been retrieved (complete recall), precision has typically fallen almost to zero.

The reader is referred to Faloutsos[2] for a survey of document retrieval methods.

Other Related Operations On Collections Of Text

In linguistic and lexicographic research, users of free-text retrieval software are more interested in finding the context in which particular patterns occur. PAT software from the University of Waterloo [3, 4] and the PADDY program[5, 6] (an earlier version of the present software) have both been used to perform this type of operation on the 533 Mbyte markup source of the Oxford English Dictionary. Linguistic operations may also include parsing, and stylistic analysis or comparison.

A second category of operation over text data is editing, including operations such as pattern substitution, document insertion and deletion, purging of documents matching

some criterion, and automatic generation of markup. These operations are almost essential in the maintenance of useful large-scale document collections.

Finally, there is a class of content analysis operations exemplified by automatic classification and automatic generation of abstracts.

Applications of Parallelism

Retrieving the relevant documents from a large collection is an inherently parallel task, whether or not the documents are scanned by humans or by computers. It is obvious that the collection can be divided into N sub-collections, each to be scanned by a different reader. If the retrieval method involves full-text scanning, it is equally obvious that the job will be completed approximately N times as fast by N readers as by one.

Parallel machines, particularly the Connection Machine, have been used to speed up document retrieval and to extend the size of the largest collection which can be processed[7, 8, 9]. However, the processing speed and RAM capacity of large parallel machines can also be used to respond quickly to changes in the document collection and/or to increase the power and expressivity of the retrieval query language. The work described here focuses on these aspects.

The Architecture Of The Fujitsu AP1000

The Fujitsu AP1000 is a distributed-memory parallel machine consisting of up to 1,024 SPARC processors (called *cells*) inter-connected by a 25 Mbyte/sec two-dimensional torus network and also by a 50 Mbyte/sec broadcast network. Each cell supports 16 Mbytes (soon 64 Mbytes) of RAM, giving a maximum total configuration of 16 gigabytes (soon 64 gigabytes). SCSI disks may be connected to all or to a subset of cells. At present, all other I/O is controlled by a Sun front-end machine called the *host*. The architecture of the AP1000 is more fully described elsewhere.[10, 11, 12]

Aims

The first aim of the work reported here was to design an architecture for a parallel document retrieval engine capable of:

1. Efficiently identifying and retrieving documents relevant to a research topic;
2. Efficiently searching for linguistic and lexicographic patterns and displaying them in context;
3. Efficiently performing systematic editing operations over a collection of documents;
4. Supporting the content analysis and advanced linguistic operations described above; and
5. Demonstrating robustness in the presence of imperfect document markup and/or non-textual inclusions.

The second aim was to demonstrate the suitability of the proposed architecture by implementing a substantial part of the design on the Fujitsu AP1000 and observing its performance.

Major Design Issues

The Role Of Inverted File Indexes

Many retrieval systems for serial machines rely on inverted file methods in order to achieve acceptable retrieval speed over large collections. Unfortunately, serial index-based systems pay a high cost in time and space to build the inverted file. Appendix 1B of the TREC-2 proceedings[13] showed that building inverted files for 2 gigabytes of the TIPSTER collection[14] on serial machines required between 4 and 100 hours of computer time and between 863 and 1500+ Mbytes of storage (before compression). It is self-evident that the more information which is preserved in the index, the greater is the storage required. It is also intuitive that the more information preserved, the richer can be the retrieval query language based upon it.

A parallel machine can operate fast enough in full-text scanning mode to obviate indexes and signatures. However, parallelism can also dramatically speed the process of building inverted files. Masand and Stanfill [9] were able to use the Connection Machine CM5 to build an inverted file for the same TIPSTER data in 20 minutes. PADRE software running on a 512-cell Fujitsu AP1000 is capable of performing the same task in a little over 90 seconds. (See below.)

In fact, whether or not to use inverted files is a question largely independent of whether processing is done serially or in parallel. In both forms of implementation, an index dramatically speeds the retrieval process but increases the difficulty of supporting powerful search terms such as regular expressions and word suffixes.

In general, it is recommended that a parallel document retrieval engine should build an inverted file if all the conditions listed below are met:

1. Search terms which can be conveniently handled using index-based searches form a large percentage of the query load.

2. Sufficient memory is available; this is of greater significance when data and index are stored in RAM.

3. The number of queries likely to be processed in the interval between changes to the document collection is large. Even with a supercomputer, index building time may overshadow search time savings on a highly dynamic collection.

4. The "downtime" required for index building can be tolerated. A commercial information bureau offering retrieval service over a highly dynamic collection may not be able to afford to suspend query processing while indexes are rebuilt.

Use of indexes for simple terms need not exclude the use of more powerful terms based on full-text scanning methods.

The Use Of Random-Access Memory

The advantages of rotating memories such as disks over random-access memories (from now on called RAM) of the same capacity are purely economic. These advantages are magnified if the storage must be non-volatile. However, rotating memories are less convenient and generally introduce significant delays.

The performance gains associated with RAM permit the use of powerful and flexible document-processing methods which would be otherwise impractical. Currently, certain

commercial and intelligence applications can justify the extra costs implied by these methods.

It is now feasible, if not easily affordable, to base a document retrieval application entirely in RAM. There are now machines with sufficient RAM to comfortably accommodate the largest document collections currently being processed by any computerised means.

It has been this author's stated contention since November 1992 [6] that the year 1997 would see the emergence of parallel machines with 4 terabytes of RAM, sufficient to hold, index and process the equivalent of all the text in a library of more than a million volumes. At the time of writing, it is possible to order a machine with 2 terabytes of RAM, though the largest system in existence has a very much smaller configuration.

In a given application, it may be found necessary to process a larger collection of documents than can be accommodated in the RAM available. In these circumstances, rotating memory can be used to augment the RAM, hopefully without sacrificing essential functionality or performance.

Applicability of Compression Techniques

Increasing the amount of data it is possible to process with a given amount of memory is particularly attractive when using a RAM-based approach. Zobel and Moffat[15] describe the use of a semi-static, Huffman-derived compression scheme which has many desirable properties for this application. It reduces space required by a large factor (greater than 3 in a quoted example) yet, unlike adaptive schemes, allows decompression to begin at intermediate points in the file.

Decompression speed is reported to be good (400 kbyte/sec on a SPARC-2) relative to alternative schemes. However, current AP1000 cells operate at only 55% of the speed of a SPARC-2, suggesting that decompression of 10 megabytes (uncompressed) using this method on such a cell would take of the order of 45 sec, though the rate reported by Zobel and Moffat may have been slowed by disk accessing. This speed is quite adequate for an index-based retrieval method, but not fast enough to be practical in a system using full-text scanning methods.

A compression scheme which allowed dramatically faster decompression, almost certainly at the expense of some memory saving, would make compression a viable option for a full-text scanning method. Even better would be a compression scheme which allowed fast pattern matching over the compressed data, without decompression.

Kent et al[16] describe a method for compression of inverted files. As inverted files require large amounts of memory, this could represent a significant saving even if the text itself were not compressed. However, the potential memory savings in a system which supports linguistic-context oriented searching will be significantly reduced by the need to preserve term position information.

Dependence Upon Document Collection Characteristics

Ideally, a document processing system will be able to operate on totally unstructured text but also able to make use of useful structure if it exists, for purposes such as identifying documents by title or headline, and selecting documents on attributes like publication date or authorname. For relevance-based retrieval, abstracting etc. the minimum markup requirement is a means of identifying the boundaries between *documents*.

The most appropriate definition of what constitutes a document depends upon the nature of the operation being performed. A document may in fact be a hierarchical

organisation of sub-documents, some of which are contextually linked to their neighbours. An ideal document processing system should be able to adapt its definition of a document according to the task at hand.

An encyclopaedia, dictionary or casebook is a single document for the purpose of addition to or removal from the document collection and from the point of view of sharing common attributes but is too large to be a useful single unit for retrieval or abstracting. For these latter purposes, each entry or case from these omnibuses might be considered to constitute a separate document. A query relating to "volcanoes" should return entries on vulcanology, rather than the entire Encyclopaedia Brittanica. Even worse, a computer generated one-page abstract of the latter is likely to be useful only as an amusement.

An ideal document processing system should also be robust with respect to "errors" in the document collection such as typographical errors, use of multiple alternative spellings or lexicographical representations, and errors in markup. It should also tolerate absence of markup and use of different mark-up standards in different sub-collections, as well as the presence of non-textual documents and attachments. The cost of great robustness is, however, likely to be very high.

The Implementation Of PADRE

Substantial progress has been made on the implementation of a parallel document retrieval engine (PADRE) for the Fujitsu AP1000. PADRE is entirely RAM-based and employs full-text scanning methods as well as inverted files. PADRE is now a useful tool for both document retrieval and linguistic research, though not all desired functionality has yet been implemented. The structure of the program is sufficiently general to accommodate all envisaged extensions.

A brief description of current capabilities follows. More complete descriptions are to be found in Hawking.[17, 18]

Overview of PADRE capabilities

Present functionality includes linguistic searches, traditional document retrieval, and basic manipulation of document collections.

PADRE supports three categories of *primary search term* (pattern):

- Literal strings (using Boyer-Moore-Gosper (BMG) string matching);
- Numeric ranges (matches strings of digits whose numeric value lies in a nominated range, eg. 1788..1901); and
- Regular expressions (using GNU regular expression code)

The result of any PADRE search operation is called a *match set* which comprises an array of pointers into the text where each pointer references the first character of a match.

PADRE also supports three categories of *compound search term*:

- Component search (eg. *pattern* **within component** *component name*);
- Proximity searches (eg. *pattern1* **near** *pattern2*); and

- Set operations, which allow pairs of match sets to be combined using difference, union and intersection operations. Set operators allow named as well as immediate operands.

PADRE allows the user to display the lexicographic context of all matches in the current match set (or just a sample of them). In addition, searches can be grouped into research *topics* and documents may be retrieved or identified according to their ranked relevance to the current topic.

Finally, PADRE includes a number of document collection maintenance operations:

- Load raw or compressed data from a variety of sources;
- Merge pairs of collections;
- Display characteristics of the current collection;
- Delete all documents matching some criterion;
- Purge all text between specified start and end markers; and
- Dump the current collection in raw or compressed format to a variety of destinations.

Note that multiple resident collections are supported. This permits additional documents to be easily merged with an existing collection. It also enables independent querying of separate collections with low context-switch overhead, a feature which might be important in a bureau environment.

PADRE Architecture

PADRE consists of a host program which runs on the host computer and a cell program which is replicated on each of the AP1000 cells. The cell programs accept commands and data from the host, carry out operations on the data and transmit results to the host. The host program manages the cells and interacts with the user. Communication between cells and between cells and the host is mediated by the standard Fujitsu message passing library.

Discipline Of Cell Free Space Management

Each cell has a fixed amount of RAM, 16 Mbytes in current models. Figure 1 shows how the memory is used. Memory requirements of *Kernel* and *PADRE* are determined when the PADRE cell program is loaded. Consequently `fss` is a fixed point. Furthermore, *Ring Buf* size is fixed and the message passing discipline described below prevents the size of *Sys Msg* from exceeding a specified limit, thus allowing the value of `fse` to be fixed at load time. PADRE is thus able to reserve all space between `fss` and `fse`, labelled as *Free Space*, for raw data, compressed data, indexes, document tables, component tables, relevance tables and match sets. The size of *Free Space* typically approximates 12 Mbytes per cell.

Each loaded document collection is distributed across the cells. The piece of a collection held by a particular cell is referred to as a *chunk*.

As shown in figure 2, *Free Space* is divided into four areas. The ensemble is managed more-or-less according to a stack discipline and so are the areas labelled *Textbase Space*

Figure 1: Top Level Allocation Of Cell Memory. *Kernel* represents the space used by the cell operating system, *PADRE* represents the space needed for the PADRE cell program and its stack, and *Sys Msg* and *Ring Buf* the space needed for message-passing bufers. Note that *Ring Buf* is not used by PADRE and may be of zero length.

Figure 2: Top Level Allocation Of PADRE Free Space. The four subdivisions of this space are described in the text and in subsequent figures.

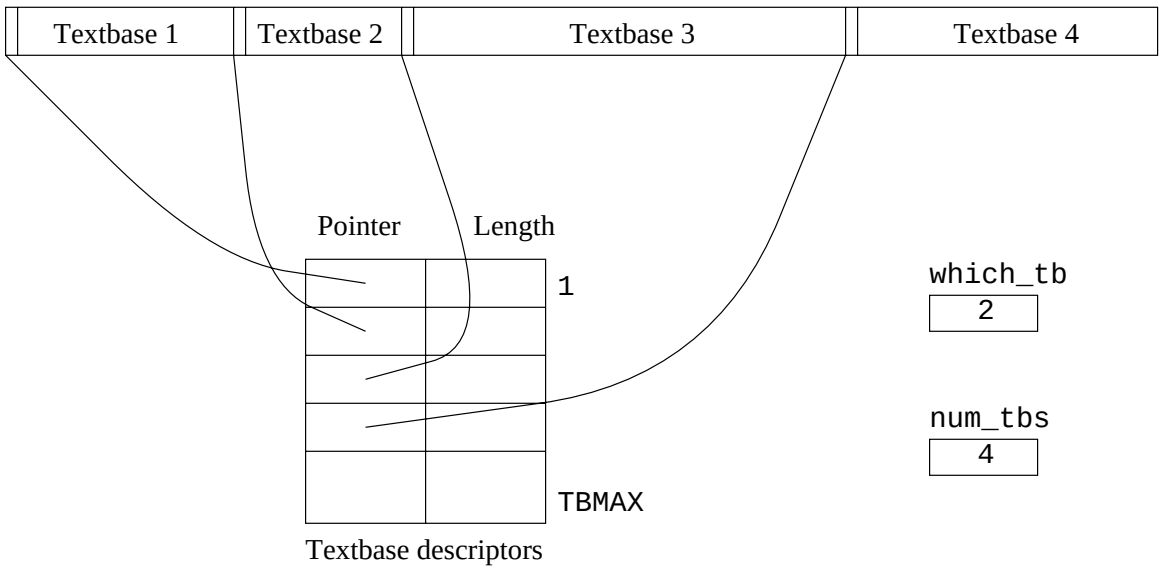

Figure 3: Example of use of *Textbase Space* when four document collections are loaded and collection 2 is current. Note that each collection is preceded by a header which includes the size of this cell's chunk and the string which is used as a start-of-document marker.

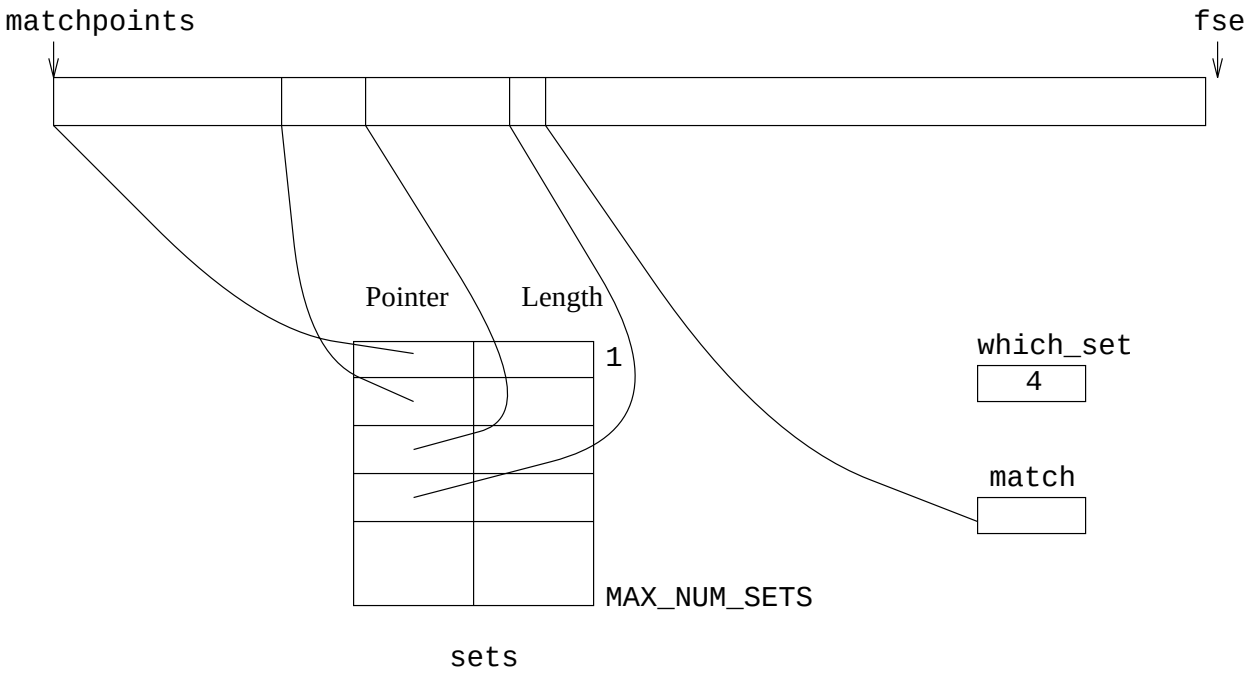

Figure 4: Example configuration of *Match-set Space*. *Match-set Space* is considered as a stack of matchpoints, shown growing to the right. Four match sets are shown, each defined by the index of the first matchpoint in the set and the number of matchpoints in the set. `Match` indicates top of stack and `sets[which_set]` indicates the start of the current match set.

and *Match-set Space*. Figure 3 shows multiple collections loaded in *Textbase Space*. If a new collection is added, the size of *Textbase Space* increases and the other areas are reset.

Doc-desc Space holds an array of pointers to the starts of all documents and a correspondingly sized array `crm` of floating-point document relevance metrics. Whenever a reset occurs, or when the user elects to use a different collection, the current document collection is scanned for document starts and `crm` is zeroed.

Table Space holds the inverted file for the current collection and, if any components have been defined, tables of component start and endpoints. If the user either requests that an inverted file index be built for the current collection or defines one or more document components *Match-set Space* will be reset and *Table Space* will grow.

Match-set Space holds a stack of match sets. Figure 4 shows a typical configuration. Each search results in a match set which is added to the top of the match set stack. Set operations and proximity operations may replace the top n items on the match set stack with a new set. *Match-set Space* is reset when any of the other areas grows or when the user anounces a new topic.

If a compressed document collection is loaded or if the current collection is to be dumped in compressed form, the *Free Space* above *Textbase Space* temporarily holds the compressed data.

Methods Of Loading Data

A document collection may be loaded from a single file on the host [5, 6, 19], from files on cell disks[19] or from hierarchies of compressed files such as those used to distribute the TIPSTER collection.

In the latter case, the file directory is recursively scanned and files which pass a simple file-type and filename filter are loaded one by one. The first file goes to cell 0, the second to cell 1, and so on with wraparound when the last cell has received its data. Once a cell receives all of a file, it starts to decompress it, decompression proceeding in parallel with input to other cells. Time to load and decompress is thus normally the sum of the i/o time plus the time to decompress the last file.

In order to simplify search and retrieval operations, PADRE prevents articles being split across more than one cell. Document collections loaded from a single uncompressed file are initially arbitrarily split into equal sized chunks but a process of shuffling[19] recombines partial documents before searching commences.

The Importance Of Load Balancing

In general, time taken for full-text scanning is proportional to the amount of data to be scanned. Furthermore, the time taken to scan a document collection distributed over the AP1000 is the time taken by the slowest cell to search its chunk of the collection. The slowest cell will normally be the one with the most data. A measure of load imbalance LI is defined as the ratio of maximum chunk length to average chunk length. Reducing it to 1.0 will not only increase speed of searching by a factor of LI but will also permit the processing of a maximal amount of data.

In general, the constraint that documents may not be split across cells prevents the achievement of perfect load balance.

Load imbalance is a significant factor in handling real world document collections in PADRE. Naive loading of parts of the previously mentioned TIPSTER collection on AP1000 configurations may result in load imbalance measures exceeding 2.0 with consequent severe degradation of capacity and performance. This collection includes at least one document of over 2 Mbytes in length. Such long documents are not only inconvenient for retrieval but may significantly impair the ability to balance load.

Implementation Of Load Balancing

The great computational cost of obtaining optimal load balance over a collection exceeding a million documents is is not justified by the performance gains likely to be achieved relative to a simpler, less optimal approach.

The current version of PADRE includes a user command to balance load. The algorithm assumes that the document collection is not ordered, allowing documents to be transferred between arbitrary pairs of cells. At present, however, it only considers documents at the tail of a chunk for transfer to another cell.

In essence, all-to-all communication is used to form a table (replicated in each cell) of each cell's chunk size. The table is sorted and cells at the top of the table pair with those at the bottom, top with bottom, 2nd top with 2nd bottom and so on. The cells in each pair negotiate a number of whole documents (possibly zero) to be transferred between them so as to minimise the maximum of the two resulting sizes. These pair-wise transfers occur in parallel.

The PADRE `loadbalance` command may be issued repeatedly until LI, displayed after each operation which might have changed it, has fallen to a satisfactory level or until the algorithm ceases to be effective.

A Message Passing Discipline

Data may pass in significant quantities between the cells and between cells and the host during the following operations:

- Loading a document collection;
- Dumping a document collection;
- Assembling complete documents;
- Load balancing; and
- Returning retrieved documents

To control the amount of memory required for system message buffers, blocks of data transmitted during these operations are broken into packets whose length does not exceed a PADRE-specified threshold. This threshold is normally set at 0.5 Mbyte but could be reduced to to say 0.1 Mbyte without significantly reducing speed. Packets after the first are not transmitted until receipt of the predecessor is acknowledged.

Address Space Limitations

In general, the number of bits reserved for character addresses in a processor or filesystem may limit the size of document collection which can be processed. Parallelisation along PADRE lines extends the limit by a factor of N by dividing the document collection into N chunks, each of which may be independently addressed. PADRE currently allows sufficient address bits to address up to 4 gigabytes per cell. PADRE also uses double-precision floating point numbers to compute and report the overall size of the collection.

Indexing

PADRE software includes the ability to build an inverted file comprising a sorted array of pointers to the first character of each indexable term.[5] It should be noted that, in the parallel implementation, there is no need to form a complete inverted file. Each cell needs only an index to its own chunk of the collection.

During retrieval operations using the inverted file, the binary search method is used to locate literal terms. If an index has been built (in response to user command), PADRE will use it in subsequent searches for occurrences of literal terms which are anchored at word starts.

Use Of Global Reduction Operators

The Fujitsu AP1000 provides hardware support for global reduction operators such as global sum, global maximum, and global minimum. These operations have been found useful in:

1. Relevance ranking across the complete document collection;
2. Reporting overall collection statistics; and
3. Calculating term frequencies across the entire collection.

Software Testing Issues

PADRE is a moderately complex system (of the order of 10,000 lines of C code) comprising two interacting programs each implemented as approximately ten modules. Complexity is increased by the necessity to use message passing.

Coding errors in the handling of special cases or boundary conditions in large data sets are likely to manifest as plausible but incorrect results with no indication of error. Accordingly, a confidence testing procedure has been adopted to ensure credibility of results and reduce the risk of introducing undetected errors when changing algorithms or adding functionality.

The testing procedure is based on an artificial data file with known checksum. The test script first checks the checksum, and replicates the data a number of times. It then runs PADRE to process a wide selection of commands over the replicated data and checks the results against a manually verified set of results. Commands and data are designed to exercise as many special cases in the code as possible.

Error handling

Any software system which is offered as a production user tool should incorporate sensible mechanisms for detecting, reporting and recovering from error conditions. Relative to a serial machine, error detection and recovery on a parallel machine is likely to introduce more complexity, and bring with it a greater requirement for user-friendly error-reporting. Use of a parallel machine opens the possibility that each of hundreds or thousands of processors may detect the same error. The designer of parallel software must avoid flooding the user with full details of every such occurrence.

In the PADRE context, errors detected in the cell program are treated specially. Such errors include:

- exceeding array limits or available memory due to loading, matching etc.,
- error returns from message passing calls,
- error returns from i/o calls to cell disks,
- errors related to characteristics of the document collection, such as unmatched component markers, missing start-of-document markers etc.

A number of techniques for reporting errors have been found useful in PADRE.

Spatial Aggregation

Some tests for error conditions are made in highly parallel sections of the code, in which all cells perform the same test. In these cases in the PADRE model, all cells call a synchronised-error reporting routine `err_cnt()` after performing a test for an error condition. They supply a code for the error, a flag indicating whether or not the error condition was detected and additional information (such as the value of the array index which overflowed) in the form of the integer.

`Err_cnt()` synchronises all cells, then uses global reduction operators to calculate the number of cells which reported the error and maximum and minimum values of the additional information. The results are passed back to the host program as a specially-typed message for reporting (with appropriate explanation) to the user.

Temporal Aggregation

Some tests for error are not synchronised across all cells. In these cases, spatial aggregation is not feasible and another mechanism is employed to prevent message bombardment. In essence, each error is reported as it occurs up to a specified number of occurrences. After this threshold is reached, errors are counted but no longer reported. On program termination, a summary of unreported errors is displayed.

Handling Cell Error Reporting On The Host

Ideally, cell-detected errors would be processed as exceptions on the host but this is not currently implemented. Instead the PADRE host program is engineered to call the `wait_for_msg()` function whenever a message is expected from a cell. `Wait_for_msg()` receives messages of all types, and handles error reporting to the user if intervening error packets are received.

PADRE Performance

Times reported in this section are elapsed wall-clock times measured by the host computer. They are measured in this way to give a realistic indication of the delays which would be experienced by a human user. Measurements taken in this way are potentially subject to considerable influence from other activity on the host machine, but the individual observations contributing to averages reported below in fact showed relatively little variability.

Time To Load Data

The bandwidth of the link between the host computer and the cell array is currently a bottleneck. Consequently, the rate of loading a document collection (using compression and memory mapping techniques) is unlikely to exceed **3.3 Mbyte/sec**. This figure is based on the lowest of ten observations of time taken to load 243 Mbytes of US patent descriptions.

A rudimentary implementation of software to load document collections from cell-connected disks exists but needs radical optimisation. Using a configuration consisting of 128 AP1000 cells, 32 of which are connected directly to a disk, an aggregate loading speed of approximately 13 Mbyte/sec has been observed. With compression, this would increase to approximately 32 Mbyte/sec. These figures are dramatically less than the peak speed of the hardware.

Potentially, each disk should be able to transfer approximately 2 Mbyte/sec. of raw data, translating to a data loading speed of 4-5 Mbyte/sec per cell if the data were stored in compressed form. An aggregate loading speed in excess of **2 gigabyte/sec.** on a maximal AP1000 configuration (512 cells with disks) may be achievable.

Time To Balance Load

The amount of time taken for a single load balancing operation depends upon the amount of data exchanged between the cells and the amount of network contention. It also depends upon the size of the largest resulting chunk, as, after load balancing, the collection must be scanned for document starts and relevance metrics for each document must be reset.

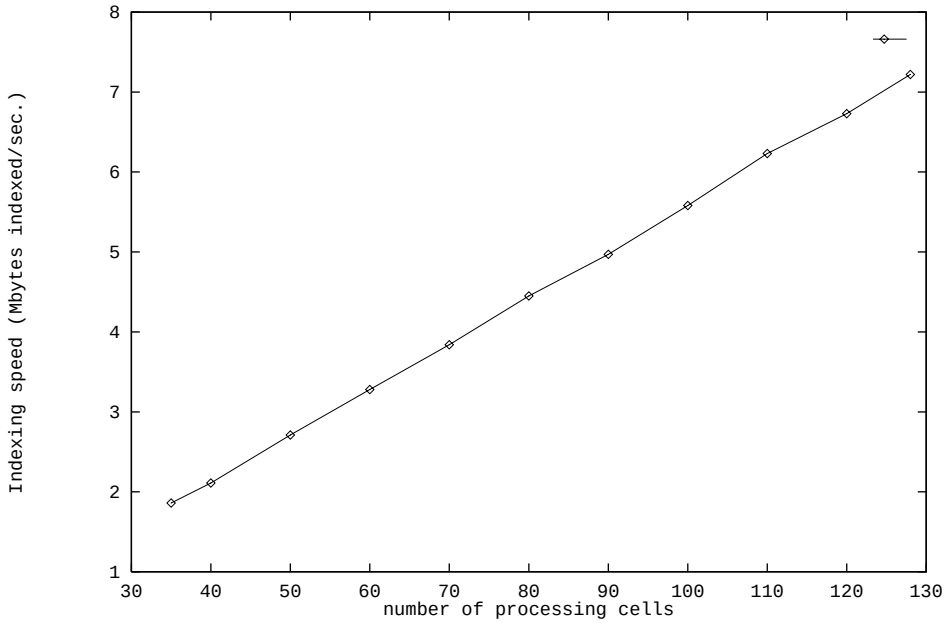

Figure 5: Rate of index building over a 243 Mbyte collection of patent descriptions as a function of number of AP1000 cells employed. Each point is the mean of five observations.

Load balancing times are small in comparison with those for loading and indexing and almost always pay off in reduced scanning times. It is likely that the longer it takes to balance load, the greater is the payoff, because large-scale data exchanges imply significant improvement in load balance. However this conjecture has not been studied empirically.

Time To Build Inverted Files

Figure 5 shows the relationship between indexing speed and the number of processing cells. As can be seen, speed up is close to linear. This may seem surprising as the time-dominant operation in building an index for a chunk is sorting. PADRE uses the well-known quicksort algorithm whose average running time is dominated by a term proportional to $n \log n$, where n is the number of word starts in the chunk, assumed proportional to the chunk size. The effect of increasing the number of processing cells is to reduce the amount of data to be indexed in each cell. One would therefore expect a non-linear relationship.

To investigate this phenomenon further, index building times for a wider range of chunk sizes were measured and plotted in figure 6. Also plotted were $m_1 n$ and $m_2 n \log n$, where m_1 and m_2 were chosen to make the plots pass through the extreme observed data point. As can be seen, the two theoretical plots do not significantly depart from each other in the relevant range and the observed data points lie between them.

Three observations were made of the time taken to index the first two CDs of the TIPSTER collection using a 512-cell AP1000. After removing manually generated indexing information, the size of the collection to be indexed was approximately 1.98 gigabytes. The total number of entries in the index was 327,222,379 and the average of the three index-building times was **92.18 sec.** The three observations differed from each other by no more than 0.01 sec.

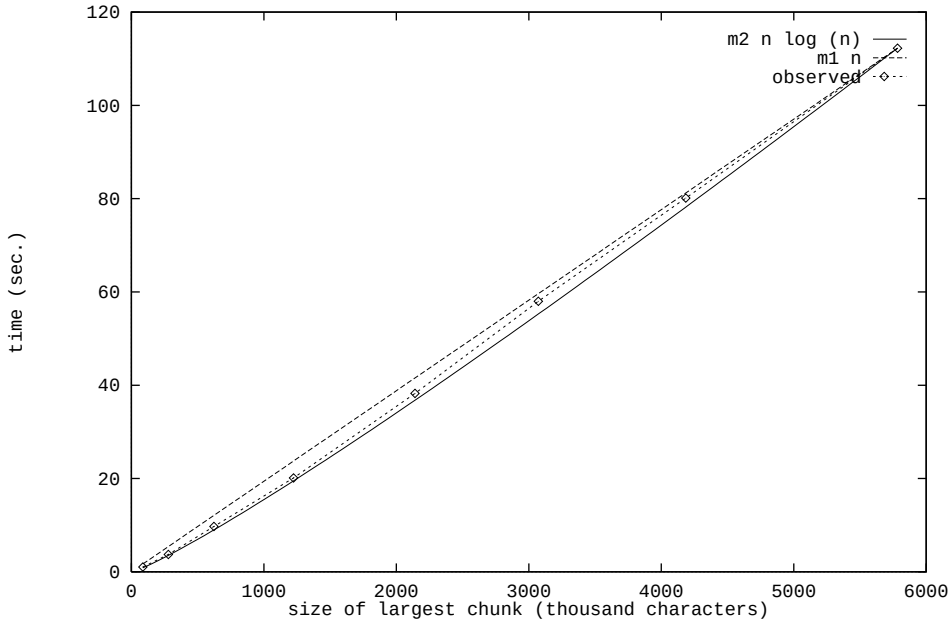

Figure 6: Time taken to build an inverted file index for eight different subsets of the TIPSTER data as a function of the size of the largest chunk. Each point is the mean of five observations. A 128-cell AP1000 was used.

Search Times

Figure 7 graphs the relationship between search time and number of processing cells for a literal term (Boyer-Moore-Gosper search) and an index-based search. As can be seen, search times for the BMG case increase as the number of processing cells decreases, whereas the indexed times appear to remain relatively constant. It is almost certain that the times for indexed searches show a similar relationship to number of cells as in the BMG case but the effect is lost due to the way times are measured. The time taken for an actual indexed search is believed to be negligible compared to overheads which are independent of the number of processing cells, such as job scheduling on the host.

To illustrate PADRE's performance when searching for non-literal terms, measurements were conducted of the time taken to search for a regular expression designed to find all words containing exactly two occurrences of the letter *i*:

```
>> regexp "\<[a-hj-z]*i[a-hj-z]*i[a-hj-z]*\>"
```

over 1.98 gigabytes of the TIPSTER data on a 512-cell AP1000. The time taken to process this query was **72.73 sec.**. This compares with times of **3.55 sec.** and **0.77 sec.** to find all occurrences of the string `walrus` using the regular expression code and the BMG code respectively. Each time quoted is the mean of five observations. The greatest observed deviation from the mean was 0.01 sec.

Responsiveness To Changes In Document Collection

In an experiment designed to measure PADRE's responsiveness to additions to the document collection, 509 Mbytes of data from the Wall Street Journal was loaded into a

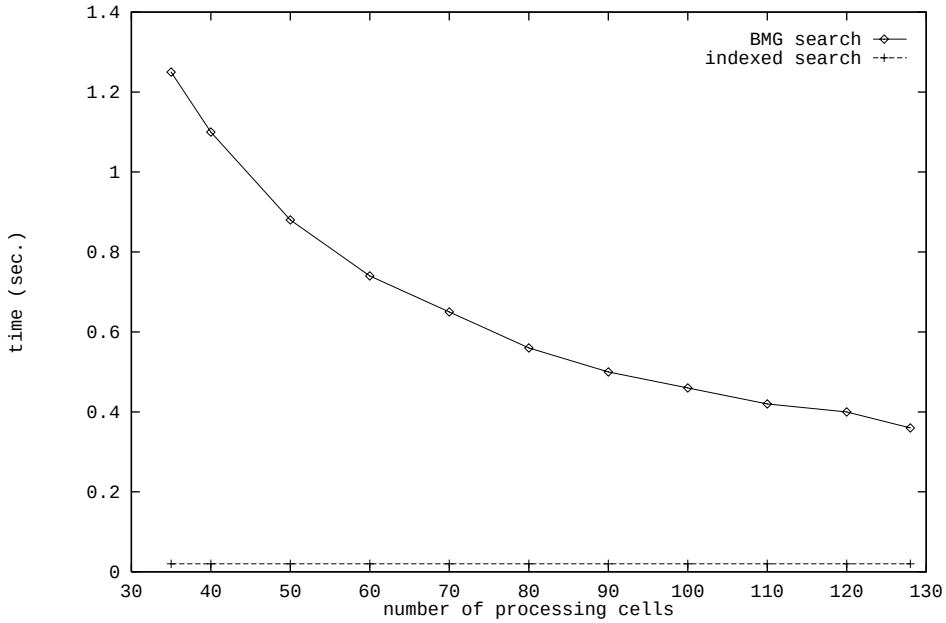

Figure 7: Relationship of search times over a 243 Mbyte subset of the TIPSTER data (US Patent descriptions) with number of processing cells. Each point is the mean of five observations.

128-cell AP1000 and used for searching. An additional 10 Mbytes of data (Associated Press reports) was subsequently loaded from a disk on the host and merged with the existing collection. The combined collection was then balanced across cells. The mean of three measurements of time elapsed between issuing the load command and the display of results of the first post-merging search was **18.7 sec.** Roughly half of the time taken was due to loading from the host.

In a further experiment designed to measure responsiveness to deletions, the interval between the issue of commands to remove all documents including the word "computer" and the completion of the next search was **9.2 sec.**, averaged over three measurements. The document collection reduced in size by 57 Mbytes.

PADRE Limits On Document Collection Size

As previously mentioned, PADRE free space on the current AP1000 approximates 12 Mbytes. Loading 10 Mbytes of documents per cell allows sufficient room for the match-sets generated by a typical complex topic, but not for an inverted file. As the largest possible AP1000 configuration comprises 1,024 cells, 10 gigabytes is a practical bound on the size of data which may be processed on a current AP1000 using PADRE. When the amount of RAM per cell increases to 64 Mbytes, this bound is expected to increase by a factor of 5, rather than 4, to 50 gigabytes because the per-cell memory overhead of kernel, PADRE code and message buffers remains constant.

PADRE has not been tested on a 1,024 cell configuration but tests using 5 gigabytes of data on a 512-cell machine have been carried out successfully. Timings for this run are tabulated in figure 1.

No. of processing cells	512
Document collection size	5,471,469,449 characters
Load time over ethernet	87 min.
Max. cell memory gained by load balancing	1.41 Mbyte per cell

Operation	Min. Time (sec.)	Max. Time (sec.)	Ave. Time (sec.)	No. Obs.
literal search	1.16	4.75	1.88	25
union	1.27	2.49	1.74	16
near 3	0.05	0.11	0.07	6
near 2	0.01	0.05	0.04	9

Table 1: Timings and sizes observed during processing of an artificial document collection 5.10 gigabytes in size using a 512-cell AP1000.

Discussion And Conclusions

Timing results presented for PADRE show that a parallel distributed memory machine permits fast retrieval operations on large collections of documents using otherwise impractical full-text scanning methods. It has been demonstrated that these methods allow simple implementation of powerful query terms and quick response to alterations in the document collection.

Implementation and experimentation to date has been restricted to the achievement of the first two goals for a document retrieval system which were listed in the aims section above. However, PADRE's ability to delete text between nominated start and end markers gives an indication that achievement of the third goal will not be difficult. It is contended that the design and basic framework of PADRE are adequate to support the achievement of all five goals, but further work is needed to demonstrate this.

The PADRE concept is readily portable to other distributed memory parallel machines such as Connection Machine CM5, IBM SP2, Intel Paragon and Cray T3D. It would be relatively easy to recode PADRE to use a portable message passing library such as MPI or PVM, without great loss of efficiency. Taking this approach would allow fairly easy porting to a possibly heterogeneous network of workstations.

However, using a loosely coupled and loosely managed network of machines makes load balancing difficult, reduces reliability and increases the difficulty of achieving the global synchronisation necessary to ensure consistent rankings over the entire collection.

PADRE is envisaged as a tool for a large-scale commercial environment in which fast searches can be made over a complete data collection subject to controlled evolution. In such an environment, authoritativeness of results and ease of management are critical, and therefore a tightly coupled machine is preferred.

Acknowledgements

Fujitsu Laboratories provided access to AP1000 machines in the Fujitsu Parallel Computing Research Facility and assisted in various ways. Robin Stanton and Paul Thistlewaite

gave support and advice. Peter Bailey and Michael Hiron worked on elements of PADRE code. GNU regular expression code from the Free Software Foundation is incorporated in PADRE. The National Institute of Standards and Technology (U.S.A.) and various copyright holders provided access to the TIPSTER data collection. I extend my sincere thanks to all these people and organisations.

Recent work on PADRE has been supported by the Co-operative Research Centre for Advanced Computational Systems (ACSys).

References

[1] D. K. Harman 'Overview of the Second Text REtrieval Conference (TREC-2)', in *The Second Text REtrieval Conference (TREC-2)* US Department of Commerce, NIST Special Publication 500-215, pp. 1-20 (Mar 1994).

[2] C. Faloutsos 'Access Methods For Text', *Computing Surveys*, **17**, (1) (Mar. 1985), pp. 49-74.

[3] H. Fawcett *PAT 3.3 User's Guide*, University of Waterloo Centre for the New Oxford Dictionary, Waterloo, Ontario, Feb 1991

[4] G. H. Gonnet, R.Y. Baeze-Yates and T. Snider *Lexicographic indices for text: Inverted files vs. PAT trees*, Report OED-91–01, University of Waterloo Centre for the New OED and Text Research, Waterloo, Ontario, Feb 1991.

[5] D. A. Hawking 'High Speed Search of Large Text Bases On the Fujitsu Cellular Array Processor', in *Proceedings of the Fourth Australian Supercomputing Conference* pp. 83-90. Gold Coast, Australia, (Dec 1991).

[6] D. A. Hawking 'PADDY's Progress (Further Experiments in Free-Text Retrieval on the AP1000)', in *Proceedings of the First Annual Users' Meeting of Fujitsu Parallel Computing Research Facilities*, paper ANU-8, Kawasaki, Japan, (Nov 1992).

[7] C. Stanfill and B. Kahle 'Parallel free-text search on the Connection Machine system', *Commun. ACM* 29, 12 (Dec. 1986), pp. 1229-1239.

[8] C. Stanfill and R. Thau 'Information retrieval on the Connection Machine: 1 to 8192 gigabytes', Technical Report DR90-3, Thinking Machines Corporation, Cambridge, Mass., 1990

[9] B. Masand and C. Stanfill 'An Information Retrieval Test-bed On The CM5', in *The Second Text REtrieval Conference (TREC-2)*, US Department of Commerce, NIST Special Publication 500-215, pp. 1-20 (Mar 1994).

[10] T. Horie, H. Ishihata, T. Shimizu and M. Ikesaka. 'AP1000 Architecture And Performance Of LU Decomposition,' in *Proc. 1991 Int'l Conf. On Parallel Processing*, pp. 634-635, August 1991.

[11] H. Ishihata, T. Horie, S. Inano, T. Shimizu and S. Kato 'CAP-II Architecture,' in *Proceedings of the First Fujitsu-ANU CAP Workshop*, paper 1, Kawasaki, Japan, (Nov 1990).

[12] T. Horie, M. Ikesaka and H. Ishihata 'Interconnection Network For Multiprocessors', in *Proceedings of the First Fujitsu-ANU CAP Workshop*, paper 2, Kawasaki, Japan, (Nov 1990).

[13] D. K. Harman *The Second Text REtrieval Conference (TREC-2)*, US Department of Commerce, NIST Special Publication 500-215, (Mar 1994).

[14] D.K. Harman 'Overview of the First Text REtrieval Conference (TREC-1)', in *The First Text REtrieval Conference (TREC-1)* US Department of Commerce, NIST Special Publication 500-207, pp. 1-20 (Mar 1993).

[15] J. Zobel and A. Moffit 'Adding Compression To A Full-Text Retrieval System', in *Proceedings of the Fifteenth Australian Computer Science Conference*, Hobart, Australia, pp 1077-1089 (Jan 1992).

[16] A. Kent, A. Moffit, R. Sacks-Davis, R. Wilkinson, and J. Zobel 'Compression, Fast Indexing, and Structured Queries on a Gigabyte of Text', in *The First Text REtrieval Conference (TREC-1)*, US Department of Commerce, NIST Special Publication 500-207, pp 229-244 (Mar 1993).

[17] D. A. Hawking *PADRE User Manual*, Department of Computer Science, Australian National University, Canberra, Australia, Oct 1994.

[18] D. A. Hawking 'Searching For Meaning With The Help Of A PADRE', *To Appear In Proceedings Of The 1994 Text Retrieval Conference (TREC-3) — In Preparation*

[19] D. A. Hawking and P. Bailey 'Towards a Practical Information Retrieval System For The Fujitsu AP1000)', in *Proceedings of the Second Fujitsu Parallel Computing Workshop* paper P1-S, Kawasaki, Japan, (Nov 1992).