High Speed Search of Large Text Bases on the Fujitsu Cellular Array Processor

David Hawking
Computer Science Department
Australian National University
GPO Box 4, Canberra ACT 2601
(dave@anucsd.anu.edu.au)

Abstract

The Australian National University has recently acquired an "experimental" 64 cell, distributed memory, cellular array processor (CAP) under a collaborative research agreement with Fujitsu Research Laboratories. Its considerable processing power and very large total amount of memory (2 gigabytes at the time of the conference) render feasible a brute–force parallel approach to free text retrieval. The brute–force approach offers great flexibility and, because it obviates the need for time–consuming index building, is well adapted to rapidly changing text bases. Very encouraging results have been achieved with a partial CAP emulation of an existing index–based serial program for dictionary research. A 550 megabyte dictionary was used as the example text base. Dramatic speed–ups have been achieved with a parallel implementation of conventional indexing techniques in which building the index is achieved in minutes rather than days. It is argued that practical systems built on CAP–like architectures could perform at impressive speeds on frequently changed text bases whose size is measured in tens of gigabytes.

1. Introduction

There are many real–world applications which require high speed searching of large collections of text for portions containing specified words, parts of words, phrases or combinations of them. Examples include searches of legal cases or statutes, searches of scientific abstracts, searches of newspaper reports and dictionary or other language research. The text to be searched may be a single document or a collection of documents. The quantity of text to be searched is typically at least a significant fraction of a gigabyte.

The work reported in the present paper has been carried out in collaboration with the Australian National Dictionary Centre (ANDC) using the markup source of the Oxford English Dictionary (OED) as the example text base. It is a single document of 550 megabytes!

The first approaches to searching large text bases only allowed retrieval of documents where the search terms (in this case always words or phrases) matched author–supplied keywords. In many applications, however, this approach is impractical or too restrictive. Later *free text retrieval* systems matched search terms against all terms in the documents.

Today, there are many free text retrieval systems in use on sequential computers which allow very rapid searches using automatically computed index files. STATUS, used in legal retrieval applications, and PAT, developed for dictionary research at the University of Waterloo, are two examples. Gonnet et al [2] review alternative types of index for this application and report on the methods used in the PAT program.

There are several disadvantages of sequential index–based approaches:

1. Index files take up large amounts of disk space,
2. Index files take a very long time to compute, with elapsed times measured in days for text bases of hundreds of megabytes. Systems relying on such indexes are consequently unable to cope with rapidly changing text bases.
3. The index is usually built in a way which precludes some profitable searches.

The advent of large memory parallel computers opened the way to more flexible search techniques. Stanfill and Kahle [5] reported in 1986 on a parallel free–text search method developed for the then fledgling Connection

Machine. They addressed the problem of how to find relevant [small] documents among a large set and used the technique of *surrogate coding* to reduce the test of whether a word was present in a document to simple logical tests. In this model of retrieval, documents are ranked for relevance according to how many of a set of search terms (generally words) defined by the searcher are found in them.

Several more recent papers (for example Stone [8], Waltz et al [9], Salton and Buckley [3], Stanfill [4], Stanfill, Thau and Waltz [7] and Stanfill and Thau [6]) have since focussed on analyzing and/or improving the performance of parallel methods for extracting relevant documents from very large collections. Stone and Salton and Buckley have disputed the claimed massive benefits of parallelism in text retrieval. Stone argued that use of inverted index files on a serial machine permitted equal or better performance than non–indexed parallel approaches. In response, Stanfill, Thau and Waltz and Stanfill and Thau have since used inverted indexes in combination with massive parallelism.

Using indexing techniques on the Connection Machine, Stanfill and Thau [6] make the attention–attracting claim that "*currently available hardware can deliver interactive document ranking on databases between 1 and 8192 gigabytes of text*"! However, ranking (finding the documents matching a set of search terms most closely) is not the most difficult part of the problem. As an estimate of the size of data which could be handled by a practical text retrieval system, 8192 gigabytes is unrealistic by two or three orders of magnitude because:

1. Stanfill and Thau assume that the inverted index could be stored in a series of Data Vaults attached to a Connection Machine CM–2. They state that a CM–2 is limited to 8 simultaneously active Data Vaults and that the largest CM–2 configuration extant has only 3. However, even the index for an 8192 gigabyte (8 terabyte) collection of documents would occupy 64 Data Vaults (about 3 terabytes)!
2. In conventional index–based approaches, the time to create the index dwarfs retrieval time by many orders of magnitude. Even if the problems associated with storing and accessing 8 terabytes of data and 3 terabytes of index can be solved, the time taken to build the inverted file is likely to be so long that documents eventually retrieved would be of historical interest only!

The aim of the present work is to investigate the usefulness of a large–memory parallel (MIMD) machine in the types of text searches needed by people writing and updating dictionaries or doing research on language. The functions of an existing sequential text search system have been emulated on the Fujitsu AP1000 using both indexed and brute–force methods and practical performance comparisons carried out.

2. Text Searching In Dictionary Writing and Language Research

The retrieval problems arising in dictionary and other language research do not conform to the model assumed by most of the papers cited in the introduction. Dictionary research does not involve the retrieval of sets of documents relevant to some topic of interest but rather the location in context of all occurrences of words or phrases of interest. It is the meaning and usage of a word rather than the meaning of a document which is important.

The Oxford family of dictionaries are available in marked up electronic form. The mark–up uses SGML (Standard General Markup Language) conventions to delineate the logical structure of the document and tag the content. It is also used to define the typeset form of the printed dictionary. A typical dictionary file consists of a list of entries starting with the tag `<e>` and ending with `</e>`. Inside an entry is a headword (preceded by `<hw>` and followed by `</hw>`) followed by a potentially long list of other components giving etymology, definitions, subject area, and supporting quotations with date, source and author information.

The Australian National Dictionary Centre uses the PAT program, described by Fawcett in [1], to assist its dictionary production. PAT runs on a Sun SPARCstation. The Centre is currently producing an Australian version of the *Concise Oxford Dictionary* and using PAT in various ways.

PAT assists in maintaining consistency by finding all occurrences of alternative forms of the same expression eg. "Air force", "airforce" and "Air Force". PAT is also used to extract all of the entries in a particular subject area (eg. `<sj>Comp. Sci.</sj>`) to be sent for vetting by subject specialists. Furthermore, PAT is used to find all occurrences of British and Imperial terms (eg. "cooker" (= stove) and "cubic furlong") prior to context–de-

pendent substitution of Australian terms. LECTOR, a PAT companion program, typesets the located entries on a screen window.

If original source material such as newspaper clippings is available in electronic form, PAT can be used to search for quotations to support a particular usage of a word.

Dictionary and language researchers may use the quotations embedded in larger dictionaries to study word usage. For example, using PAT, it is a trivial matter to find all words with a Hebrew etymology and not too difficult to find all the quotations dated between 1500 and 1750 which include the words “shall” or “will”.

Before searches of the dictionary of interest can be made using PAT, the dictionary must be processed by a companion program (PATBLD) to produce a tree–structured index. The index–building process is very time consuming and the result is an index file of comparable size to the original dictionary. To save time and disk space, the dictionary is normally indexed only at word starts. *Stop words* such as “and” and “the” are normally excluded.

With the appropriate index files in place, the dictionary researcher can use PAT to search for patterns or combinations of patterns either in the whole range of the text or in components of it delineated by SGML commands or other markers. PAT records all the matchpoints encountered as a numbered set and reports the number of matches. The user can then request that all (or a sample) of the matches be printed out with a specified number of characters of pre and post context. The default mode is that the matches are printed in lexicographic order which means that, if the appropriate pattern is chosen, PAT can generate full or partial concordances of the text.

PAT can also be asked to find the most frequently occurring words or phrases in the text (**signif** command) and to find the longest repetition (**lrep** command) in the text.

3. Search Commands Supported by PAT

Basic Search Patterns

a. A literal string. All words starting with the string are considered matches. Eg. “gun” will match *gunboat*, *gun*, *gunwhale* etc. To match just *gun*, the search pattern must be “gun ”.

b. A lexicographic range. All words starting with a sequence of characters lexicographically within the defined range are considered matches. Eg. “gua”..“guz” will match *gun*, *gut*, *gulley*, *gunboat* etc.

c. A numeric range. All “words” starting with a sequence of digits numerically within the defined range are considered matches. Eg. “1897”..“2001” matches *1898*, *1924*, and *1927034*.

Complicating factors

d. a blank in a pattern is taken to match any sequence of word–separator characters and stopwords. Eg. “wind rain” matches *wind and rain*

e. characters in the dictionary may be mapped to other characters for the purposes of matching but always appear unchanged when matches are printed. For example, control characters are mapped to spaces and case folding is applied during matching.

Combinations of Patterns

f. The **near**, **not near**, **fby** and **not fby** commands can be used to search for combinations of patterns. Eg. “hat ” **near** “coat ” matches all occurrences of *hat* that are preceded or followed by *coat* within a proximity range (80 characters by default).

Search Within Components

The keywords **within** and **including** are used to restrict searches to a particular type of component such as a quotation or an etymology.

g. “rhubarb ” **within docs** Q
matches all occurences of *rhubarb* within quotation components.

h. **docs** Q **including** “rhubarb ”
matches the beginning (ie. <qt>) of all quotation components including *rhubarb*

Set Operations

The list of matches produced by a PAT search is called a set. PAT provides means to name sets and to produce the union, difference or intersection of two sets.

Previously generated results can also be used in component search commands. In the following example taken from the PAT manual [1], ”%” is used to refer to the immediately preceding results. The aim of the series of commands is to find the names of dictionary entries which include quotations including the word science:

i. >> docs Q including “science ”

j. >> docs E including %

k. >> docs HL within %

4. Status of the Parallel Implementation

The Fujitsu AP1000 program PADDY emulates many of the functions of PAT and serves as an experimental vehicle for idea testing and performance comparison. In most cases of unimplemented functionality it is easy to see how the features could be implemented and easy to predict the performance of the parallel version. However, it is not yet clear how to emulate the PAT command which finds the longest repeated section of text. In addition, while it is easy to find ways to emulate the PAT default mode of presenting search results in lexicographic order, finding which is optimum in some sense is an interesting direction for further work.

5. How PADDY is Implemented on the Fujitsu AP1000

The Fujitsu AP1000 (also known as the CAP, short for Cellular Array Processor) consists of an array of processor cells, each with a SPARC integer unit, Weitek floating point unit and 16 megabytes of RAM. The cells are connected in a 2–dimensional torus mesh and also to a global broadcast network and a synchronization network. Both the torus network and the broadcast net have high bandwidths.

At present, the CAP cells have no i/o capability except for the networks described. There is no shared memory. Communication with the external world is via a VME interface in a Sun 4/390 front–end.

The dictionary searching problem is inherently parallel. The text to be searched is broken up into clearly delineated entries and there is no requirement for matches to cross boundaries between entries. Consequently, the processors of the CAP can search their own chunk of the text base without needing to communicate with their fellows.

The PADDY system consists of a host program running on the front end and a cell program which is broadcast to all the cells by the host program. Both programs are written in C with calls to the CAP communication libraries. The host program accepts user commands and, having interpreted them, passes instruction messages over the broadcast network to the cells and receives messages in reply.

Loading The Text Base

If the text base (dictionary or dictionaries) comes from disk, it is memory mapped into the host program virtual address space, and arbitrarily divided into equal sized chunks which are each sent to a different cell using a library call. If the text file comes from a tape device (Exabyte), it must be read in using read() calls. The chunk size when the full OED (550 megabytes) is processed on 64 cells is about 8.6 megabytes. To limit buffer space requirements, large chunks are sent in multiple messages of 0.5 megabytes or less.

Once each cell has its chunk of dictionary data, the host broadcasts a message to all cells, asking them to check whether they have an incomplete entry at the beginning of their chunk and, if so, to forward it to the cell on their left. It is obviously faster to do this than to have the host scan the dictionary for entry boundaries as it is read in.

An unexpected problem arose when testing PADDY on a small (5 megabyte) section of the dictionary. Some entries in the OED are over 250 kilobytes (40,000 words!) long, larger than 5/64 megabytes, causing some cells to end up with no data!

If a production version of PADDY were to work on relatively static data, considerable loading speed–up would be achieved if each cell's initial data were stored in a separate compressed file on disk. The decompression within the cells should take only a fraction of the time currently taken to transfer the data from disk to front–end and from front–end to the cells.

Storage Allocation Within Cells

Each cell has 16 megabytes of memory, of which less than 2 megabytes are needed for the CAP cell kernel, the PADDY cell program, buffers and stack space. In the current implementation of PADDY, 14 megabytes are allocated in each cell for one or more dictionaries and for saving the successful matches (as pointers).

Searching For Literal Strings

The search for literal strings has been implemented in two different ways. The first uses the most straight–forward brute–force algorithm possible. Using this method, the searcher can choose whether or not searches should be bound by the PAT constraint that matches start at the beginnings of words. Relaxing the constraint allows searches for suffixes, an extension which would have been appreciated by the Dictionary Centre when they were replacing *–ize* endings with *–ise* throughout the Concise Oxford Dictionary.

The second search method relies on the existence of a word–start index whose creation and use is described immediately below. It is far faster but less flexible.

Creation and Use of A Word Start Index

If very fast searches for literal strings are required, the searcher can request that PADDY build an index of word starts for the dictionary in question. Each cell scans its text chunk and creates a table of pointers to all the word starts. These pointers are then sorted (using quicksort) so that they reference the words in lexicographic order. All occurrences of a particular search string which begin a word can thus be found by a binary search of the ordered pointer table followed by a linear scan.

The discussion of PAT arrays in Gonnet et al. [2] seems to imply that the same index building and access techinques are now used in the PAT program. A question of interest is how effectively the AP1000 (and machines of similar architecture) can be used to speed up the creation of indices to be used later by searching software running on a sequential machine.

Large amounts of memory are needed for the pointer tables. The Oxford English Dictionary (550 megabytes total size) contains 66 million words, not including SGML tags, and generates 270 megabytes of pointer tables.

Searching For a Lexicographic Range

The range is converted to a set representation. Each character position in the range can be represented as a 16–byte set, assuming a 7–bit character representation. Non–matching characters are eliminated quickly using a 4–byte (int) set which is the logical **or** of the four 4–byte components of the full set.

Searching For a Numeric Range

The lexicographic range machinery is used to build a temporary set of all strings in the dictionary of the desired length which contain only digits. The strings are then converted to integers and numerically compared to the specified lower and upper limits.

Defining Components and Searching Within Them

PAT allows two types of components: those defined at index building time and those defined by the searcher. PADDY implements only the latter. When a component is defined, PADDY uses its standard matching code to create two corresponding arrays, the first containing pointers to the start–of–component strings and the second pointers to the matching end–of–component strings. Searches for strings within components are implemented as a series of calls to the matching routine, using the relevant pointer pairs to bound the search.

More General Pattern Matching

It is intended that the freely available GNU regular expression code will be incorporated into the PADDY framework, but, at the time of writing, this had not yet been done.

Sorting of Matches

In the absence of a word–start index, the matches are found by the cells in the order in which they appear in the dictionary. Using a word–start index means that matches are alphabetised within a cell but in essentially random order as far as the whole dictionary is concerned. The "best" way to sort a set of matches distributed over the array of cells is not yet known. Dynamic method selection is likely to yield best results because some potentially fast methods may run out of memory in certain cases.

6. Performance Measurements

The timings reported for PADDY are those seen by the AP1000 front–end. The timer on the front–end is started as soon as the user command is recognized and stopped when the front–end is notified of completion. Times reported are generally slower than the best possible due to other users' activity on the timeshared front–end.

Time To Load The AP1000

The elapsed time taken to read the dictionary file into the cells of the AP100 is, of course, heavily dependent upon the load on the front–end file system. Under time–sharing the time taken to load the full OED is typically about 15 minutes or about 0.61 megabytes/sec. It seems more relevant to speak of the fastest observed time rather than average times. The fastest observed loading rate is about 1.13 megabyte/sec.

Brute–Force Searching

Search Command	**Meaning**	**No. of Matches**	**Elapsed Time (sec.)**
"t" (PAT mode)	words starting with the letter t	5,778,452	11.08
"t"	all occurences of the letter t	25,273,155	11.18
"supercomputer "		1	15.53
"goaaa".."gozzz"	all words starting with go and then any other three letters.	44,055	20.54
1788..1901	all sequences of digits whose first four digits make a number in the specified range.	1,040,018	18.80
Q=docs "<q>" .. "</q>"	defining a quotation component Q delineated by the specified start and end strings.	2,435,559	24.73
dog within docs Q	all occurrences of the word dog within the quotation components defined above.	6,616	8.46
docs Q including cat	all quotations which include the word cat	13,104	8.38

Figure 1: Search times on the Oxford English Dictionary (550 megabytes) using brute–force methods on a 64 cell Fujitsu AP1000. Observed times show little variability. Figures given are the mean of five observations.

Relationship of Speed to Number of Cells

Number of Cells	**8**	**16**	**32**	**64**
Shortest time to load text base (sec.)	64.9	62.9	65.5	62.3
Time to find all occurrences of "computer" (123 of them)	7.72	3.94	2.03	1.07

Figure 2: Variation of search and load times for a 50 megabyte dictionary subset with number of processor cells. Search times are the mean of five observations.

The data presented in Figure 2 shows that searches on 64 cells are 7.21 times as fast as those on 8 cells. It is unclear at present why the relationship between speed and number of cell is not even closer to linear.

Searching Using Word–start Index

Time to build index	Time to find all occurrences of "antimony" (327 matches)	Time to find all occurrences of "the" (2,578,161 matches)
–	10.95	11.22
176.28	0.01	0.33

Figure 3: Time to find all occurrences of a literal string in a 550 megabyte dictionary with and without use of word–start index. Times are the mean of five observations.

The speedup factor gained by use of the index ranges from about 30 to over 1,000. As can be seen, the speed of indexed matching depends heavily on the number of matches found.

Comparison with PAT

Not having access to the source code, PAT searches must be timed by hand. Using a 45 megabyte section (parts 4.1–4.9) of the OED on an 8 megabyte Sun SLC, initial PAT search times were slower than 64–cell brute–force PADDY (eg. "goanna" – 2.9 sec, "junk" – 1.6 sec) but after a while they speeded up to the point of being virtually impossible to time manually. This may be due to sections of code or index having been paged in. Searches for ranges were also very quick: ("0000".."9999" – 2.1 sec., "goanna".."goanna" – too quick to time). On the other hand, PAT seems to slow down when searching within components whereas PADDY speeds up.

It is unknown how PAT search times are affected by the size of the dictionary nor by the size of memory on the computer. At the time of writing, the ANDC did not have an index file for the full OED.

7. Discussion

PADDY brute–force searches of the whole OED (550 megabytes) take of the order of 10–12 seconds on a 64 cell CAP. This would almost certainly be slower than an index–based serial system, but I have no comparative figures. In any case, the PADDY search times fall within a loose definition of "interactive" and can be improved if the amount of text per cell is reduced. (Just add more hardware!)

The performance of the basic brute–force string–matching code as implemented so far is certainly sub–optimal. No effort has been made to hand optimize the code and cleverer algorithms have not yet been employed. The GNU regular expression code which it is planned to incorporate into PADDY uses the Boyer–Moore–Gosper speed–up when searching for literal strings.

The brute–force parallel version also offers great flexibility in search. Extension of Paddy to handle regular expressions should be straight–forward. Gonnet et al. [2] describe how regular expression matching can be supported over PAT trees but no such facility is incorporated in the production version of the PAT program.

It is likely that loading the raw text in the parallel implementation would be three orders of magnitude faster than building an index on a serial machine. From this, it is clear that the parallel approach is greatly superior if the text base is subject to frequent update.

PADDY's searching performance when word–start indexes are used is very impressive and certainly fast enough to support queries from a large number of concurrent users. It may be that the bottleneck in such a time–shared query service would be the ability to pass the results of searches across the i/o link between the Fujitsu AP1000 and the front–end Sun. Performance would be improved in this regard if results were sent as pointers into the text base on disk rather than as 80 character context strings.

The task of building n partial indexes is considerably easier than building a single complete index. When there are n separate processors and both the index and the raw data can be kept in primary memory, a massive speed up is achieved. It is likely that the index–build time for a text base of 8 megabytes per cell could be brought down to about one minute, preserving the speed of index–based searching while at the same time allowing the text base to be frequently updated. Further work needs to be done to extend the types of searches which are index–assisted.

The work reported in this paper suggests that the Fujitsu AP1000 could be used for brute–force, in–memory searches of textbases up to about 9 megabytes times the number of processor cells. The maximum configuration

AP1000 comprises 1,024 cells, implying that 9 gigabytes is the practical limit on text base size. Such a database could be loaded in about 2.5 hours in raw form or in about 50 minutes using data compression techniques.

An obvious extension to the AP1000 architecture would be the addition of local disks on some or all of the cells. In the unlikely case of a disk per cell, the time to load a 9 gigabyte text base into a 1,024 cell CAP could be as little as 10 seconds! Such disks could also be used to increase the limit on size of text bases. If indices were kept in memory and the raw text on disk, the limit could be increased to 20 gigabytes.

The vision of a practical system which, after loading 20 gigabytes of text and indexing it in a total of about three minutes, allows searches to find all occurrences of a word in less than a tenth of a second is still capable of impressing some of us even in this technology–blase world!

Acknowledgements. I am indebted to Dr W.S Ramson and Ms H.A. Michell of the Australian National Dictionary Centre for welcoming me into the centre and teaching me all I know about production of dictionaries. The work reported has also benefited considerably from discussions with other members of the ANU Computer Science Department, particularly Prof. R.B. Stanton, Dr. C.W. Johnson, Dr. B.D. McKay and Mr. R. Cohen.

REFERENCES

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

2. Gonnet, G.H., Baeza–Yates, R.A. and Snider, T. *Lexicographical 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

3. Salton, G. and Buckley, C. Parallel text search methods. *Commun. ACM 31*, 2 (Feb. 1988), 202–215.

4. Stanfill, C. *Parallel computing for information retrieval: Recent developments.* Technical Report DR88–1. Thinking Machines Corporation, Cambridge, Mass., 1988

5. Stanfill, C. and Kahle, B. Parallel free–text search on the Connection Machine system. *Commun. ACM 29*, 12 (Dec. 1986), 1229–1239.

6. Stanfill, C. and Thau, R. *Information retrieval on the Connection Machine: 1 to 8192 gigabytes.* Technical Report DR90–3. Thinking Machines Corporation, Cambridge, Mass., 1990

7. Stanfill, C., Thau, R. and Waltz, D. *A parallel indexed algorithm for information retrieval.* Technical Report DR90–2. Thinking Machines Corporation, Cambridge, Mass., 1990

8. Stone, H.S. Parallel querying of large databases: A case study. *Computer 20*, 10 (Oct. 1987), 11–21

9. Waltz, D., Stanfill, C., Smith, S. and Thau, R. *A parallel indexed algorithm for information retrieval.* Technical Report DR87–3. Thinking Machines Corporation, Cambridge, Mass., 1987