PADDY's Progress (Further Experiments In Free-Text Retrieval On The AP1000)

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

February 19, 2007

Abstract

PADDY is an experimental free-text retrieval engine for the Fujitsu AP1000. Considerable development of PADDY has occurred since the work reported at the second CAP Workshop. Search patterns now include generalised regular expressions, set operations may now be applied to the results of searches, and proximity operators have been implemented. The speed of literal searches has been increased and the time to load a dictionary from the front end has been dramatically speeded up. The scalable performance of PADDY, coupled with likely developments in the technology of AP1000-like machines, leads to a discussion of the role of a "super" free-text retrieval engine in libraries of the future. Current PADDY research now centres on use of the AP1000 to rapidly build a complete index to support subsequent text searching on a serial workstation, and on the very large sorting operation underlying it.

1 Introduction

The work reported here and in [3] constitutes a continuing investigation of the potential of machines like the Fujitsu AP1000 in free-text retrieval applications involving very large and potentially dynamic text bases. The 533 Mbyte SGML markup source of the Oxford English Dictionary (OED) is used as a sample text base.

Implementation is in the form of a C program called PADDY whose use is described in [4] and whose functions are based on those of the PAT index-based program for serial machines (Gonnet et al [2] and Fawcett [1]).

The distinguishing characteristic of the PADDY system is that the entire text base is divided into equal sized pieces and loaded into the memory of the

AP1000 processing cells. The cells then engage in a shuffling process to ensure that the data held by each cell starts and ends at points which correspond to logical divisions in the text. In the case of the OED, the logical divisions correspond to dictionary entries but in the case of news reports, beginnings and ends of articles would constitute sensible boundaries. The data held by a cell after shuffling is called a *chunk*.

2 More Flexible Search Capabilities

Two further categories of PAT operators have been implemented since [3]. In addition, full regular expression matching, not available in PAT, is now built in to PADDY.

2.1 Proximity Searches

PADDY (after PAT) implements the so-called proximity operators: *fby* (followed by), *not fby*, *near* and *not near*. The format is:

```
<pattern1> <proximity operator> <pattern2>

eg.  "Fujitsu " near  "computer "
```

In the latter case, a match is counted if the word `Fujitsu` is found within proximity, either forward or backward, of the word `computer`, where proximity (p) is a specified number of characters. If the operator is *fby* rather than *near*, only forward proximity counts. If either of the two is negated, a match is counted only if there is no occurrence of the second pattern within p characters of an occurrence of the first.

The implementation of these operators is, of course, straightforward except at the beginning and end of each cell's chunk. In practice, proximity searches are used to search for two patterns appearing in the same context. Values of p are therefore likely to be small; an occurrence of a pattern a million or so characters after another is unlikely to imply a close relationship between the two! It can be assumed, therefore, that p is less than the smallest chunksize and implementation is correspondingly simplified.

In the case of *fby*, cell c, $(c \neq NCELS - 1, c \neq 0)$ scans from the beginning of the chunk to within p characters of the end. For each occurrence of *pattern1* in this section of text a search for *pattern2* is made in the next p characters. The scan of the last p characters proceeds in similar manner, except that a linked list is created of occurrences of *pattern1* and searches for *pattern2* are restricted to the remaining characters of the chunk. If no occurrence of *pattern2* is found, a message is sent to the cell to the RIGHT, requesting a search for *pattern2* in the first n characters of its chunk.

Cell c then performs all the searches requested by its LEFT neighbour and sends answers. Finally, it matches replies arriving from the RIGHT with ele-

pattern	proximity	no. of matches	elapsed time (sec.)
`bmg "the "`	-	2170543	2.07
`bmg "this "`	-	100829	1.91
`bmg "abb "`	-	68	1.83
`"the " fby "this "`	10	136	2.28
`"the " fby "this "`	100	41772	2.85
`"the " fby "this "`	1000	372483	8.43
`"abb " fby "this "`	10	0	3.65
`"abb " fby "this "`	100	0	3.66
`"abb " fby "this "`	1000	5	3.74

Table 1: Speed of Proximity Searches on 128 cell AP1000. Each time is the average of 5 observations.

ments of the linked list and records matches associated with any positive answers.

The implementation of *near* requires an additional special phase in which cell c scans the first p characters of its chunk and requests its LEFT neighbour to search for *pattern2* in the last n characters of its chunk.

Other strategies for the implementation of these operators (eg. *fby*) are possible. One alternative would be for each cell (except the first and the last) to send its first p characters to the RIGHT and its last p characters to the LEFT and let each cell perform all its searching locally. In general, this scheme requires more memory and more memory copying, but would win out where p was small and the number of occurrences of *pattern1* was large.

A variation on the scheme implemented would only ever send a single request to a neighbouring cell, specifying the number of characters in which to search for *pattern2*. Responses to these requests would include all the occurrences of *pattern2*. How much the consequent reduction in communication time would improve overall performance is not known.

Table 1 shows the increase in time taken for an *fby* operation as the proximity parameter is increased. Two cases are shown, one in which *pattern1* occurs very frequently and one where a different *pattern1* of the same length almost never occurs.

2.2 Set Operations

Examples of the set operations supported by PADDY include:

```
S1: >> cat + dog
```

Union: S1 finds all patterns starting with `cat` or `dog`.

```
S2: >> cat fby dog ^ cat fby bird
```

Intersection: S2 finds all occurrences of `cat` that are followed by an occurrence of `dog` and an occurrence of `bird`.

pattern	match1	match2	matches	time (sec.)
`"cat " + "dog "`	2207	4375	6582	0.02
`"cat " - "dog "`	2207	4375	2207	0.02
`"cat " ^ "dog "`	2207	4375	0	0.02
`"cat " fby "dog "`				
`^ "cat " fby "bird "`	26	2	0	0.02
`"comp" - "computer"`	73099	1581	71518	0.07

Table 2: Speed of Set Operations on 128 cell AP1000. Each time is the average of 5 (elapsed time) observations. Match1 and match2 are the number of members in the operand sets. Proximity was 50 in *fby*.

```
S3: >> comp - computer
```

Difference: S3 finds all matches starting with `comp`, but not starting with `computer`.

The result of a PADDY search is represented as a sequence of pointers in an array of matchpoints. These sequences are in order of occurrence in the text base. Two entries in a table define the beginning and end of the sequence, known as a set. Accordingly, the implementation of union, intersection and difference operators requires only simple manipulations of these sequences of matchpoints.

Intersection and difference are particularly easy because the result set may be written over the top of one of the components, whereas the implementation of union relies on creation of a temporary set in order to preserve order.

Table 2 records the very small times taken to perform the various set operations. The times reported show only the time for the set operation itself, not the component searches.

2.3 Regular Expression Matching

Increased flexibility of search patterns is one of the major benefits of the non-indexed searches supported by PADDY. To illustrate the potential in this direction and to examine the time cost of exploiting it, the GNU Extended Regular Expression Matching and Search Library from the Free Software Foundation has been incorporated into PADDY. Some examples follow:

```
R1: >> regexp \<[a-z]*ize\>
```

R1 searches for words ending in *ize*, for example *size* and *privatize.*

```
R2: >> regexp \<[a-z]*consider[a-z]*\>
```

R2 searches for words with *consider* in the middle, for example *considering* and *inconsiderate.*

```
R3: >> regexp \<[a-hj-z]*i[a-hj-z]*i[a-hj-z]*i[a-hj-z]*\>
```

pattern	no. of matches	elapsed time (sec.)
`regexp "\<ampersand\>"`	5	5.61
`"ampersand "`	5	2.98
`bmg "ampersand "`	5	1.32
R1	94771	57.69
R2	14733	57.62
R3	770673	52.83
R4	477	45.88

Table 3: Speed of Regular Expression Matching On 128 cell AP1000. Each time is the average of five observations.

pattern	matches	SPARCstation SLC	AP1000
R4	117	1206.4	10.70

Table 4: Regular Expression matching over 20 Mbyte *Concise Oxford Dictionary*: Comparison of SPARCStation SLC with 128 cell AP1000. (Elapsed times in sec.)

R3 searches for words with exactly three *i*'s, for example *participation* and *indistinctly.*

```
R4: >> regexp <hw>([a-z]?)([a-z]?)([a-z]?)([a-z]?)([a-z]?)([a-z]?)([a-z]?)

([a-z]?)([a-z]?)[a-z]?\9\8\7\6\5\4\3\2\1</hw>
```

R4 searches for pallindromic headwords of up to 19 characters.

The Regular Expression Library provides a number of tools which are used by the cell program to compile the regular expression into the representation of the appropriate finite state machine and then execute it repeatedly until the end of the data chunk is encountered. The role of the host program is merely to pass on the regular expression to the cells and receive the results.

The time taken to match patterns increases with their complexity, as can be seen in table 3. However, the time is dramatically reduced when compared with the time to perform them on a workstation as seen in table 4!

In PADDY, regular expression matches are not permitted to cross chunk boundaries. This restriction greatly simplifies implementation but does not greatly reduce usefulness. It is hard to think of any practical application in which search results would cross the logical boundaries corresponding to the chunk boundaries. Implementation of a generalised *pgrep* (parallel grep) operation in which matches could span cell boundaries of the text base would be a satisfying challenge but no practical motivation for it has yet been identified.

pattern	no. of matches	old method	BMG
`a`	8601743	2.68	2.83
`do`	189554	2.50	1.47
`cat`	20429	2.47	1.10
`rake`	1353	2.48	0.91
`train`	6342	2.57	0.78
`eighth`	488	2.61	0.68
`Hawking`	208	2.41	0.58
`standoff`	13	2.51	0.52
`procedure`	1068	2.43	0.57
`crocodiles`	62	2.47	0.53
`antidisestablishmentarianism`	1	2.54	0.26
`"a "`	4878206	3.16	4.33
`"train "`	2920	3.04	1.74

Table 5: Comparison of BMG and straightforward pattern matching methods for literal patterns. (Elapsed times in sec.)

3 Speed Improvements

Significant speedups have been achieved in loading the dictionary from the host and in carrying out brute-force searches for literal strings.

3.1 Boyer-Moore-Gosper speedup of literal searches

The Free Software Foundation distributes an implementation of the Boyer-Moore-Gosper (BMG) algorithm with the regular expression library referenced above. It too has been incorporated in PADDY, giving rise to quite dramatic speedups for literal searches.

In the straightforward literal pattern matching previously used in PADDY, every character in the text chunk is compared with one or more characters of the search pattern. The BMG algorithm gains speed by reducing the number of character comparisons. It does this by laying the search pattern (of length l) across the first l characters of the chunk and comparing characters backward from the end of the pattern. Whenever two compared characters are unequal, the pattern is slid forward a distance d , where d is looked up in a table pre-computed for the particular search pattern using the non-matching character as an index. In the best case, where no character in the pattern occcurs anywhere in the chunk, speed-up by a factor of l is achieved because $d = l$ in every case.

Table 5 compares the speed of the old method with that of the BMG. The speedup in the case of the long search patterns is as expected and the fact that BMG is slower for strings of length 1 is presumably due to the cost of setting up the pattern table.

operation	elapsed time (sec.)	Mbyte/sec
normal load	883(2)	0.60
normal load (with *msg_alloc*)	445(4)	1.2
load compressed files and decompress	172(15)	3.1
compress and write files	234(4)	2.3

Table 6: Speed of Loading OED (533 Mbyte) into 128 cell AP1000. Times reported are the minimum of the number of observations shown in parentheses. The normal loads (without *msg_alloc*) were recorded on a 64 cell AP1000 but the number of cells has little effect on load time.

3.2 Loading Data Via The Host

The text base used in these experiments is stored on a SCSI disk on the host machine. The long time, typically 15 minutes elapsed, to load the data into cell memory has been an inconvenience to all concerned when PADDY is being debugged, experimented with or demonstrated. Two recent developments have dramatically reduced the load time.

First, read buffers have been allocated in the CBIF memory, rather than in the memory of the host, using the CELLOS 1.2 *msg_alloc()* call. This has the effect of avoiding unnecessary memory copying, but more importantly, it dramatically reduces the number of context switches and the amount of interprocess communication between CAREN and the host process. Double buffering is used. A zero count of the number of times the disk reading has to wait for the next buffer to empty confirms that disk reading is the rate-determining process.

Second, because the data used in the current series of experiments is static, the ability to load compressed files has been built into the current version of PADDY. To use this function, the user loads the original uncompressed dictionary and allows the cells to break it into chunks at appropriate boundaries. The user then issues the compress command which causes the cells to compress their chunk of data, and when ready, send it back to the host for deposition in a files called `cell0, cell1, ...` in a user-specified directory.

The *cload* command requires the specification of a directory containing compressed chunk files and loads the files one at a time into the appropriate cells. Once all compressed data has been loaded, cells are instructed to decompress their data in parallel. Once decompression is complete, PADDY may be used in the normal way.

The total space required to store the compressed cell files amounts to 202 Megabytes compared to the 533 for the uncompressed OED, a compression ratio of 2.6.

Times for compress and cload operations are reported in table 6. The significantly longer times for normal loading are included for comparison. Of course, speed of loading would not be a problem if disks were connected directly to the cell array.

4 Using The AP1000 To Build Index Files For Later Serial Searches

As previously reported in [3], PAT's searching performance is very good but the time taken to build the index file on which it relies is extremely long (some days) for a large text base. It is planned to address this difficulty by using the capabilities of the AP1000 to rapidly build an index capable of being used by a PAT-like serial program.

The index file consists of a list of pointers to word starts in the text base which has been sorted so that the strings pointed to are in lexicographic order. For efficiency during the index building phase, at most s characters are compared when sorting. Searches using the E-STREET serial program will rely on a binary search of the indexfile to locate the first potentially matching string followed by a linear scan until the string pointed to no longer matches the first $min(s, l)$ characters of the pattern where l is the length of the pattern.

Implementation of the index-building IDX program is well advanced, but unfortunately no results are yet to hand.

In IDX, data is loaded into all cells except cell 0, whose sole function is that of performing an ($n - 1$) way merge of the sorted partial index files created by the other cells.

When cell 0 receives the merge command, it requests all cells to send two *boats* of data for merging. Each boat contains k pointers (indices within the total textbase) and the corresponding k strings (each of length s) to enable comparisons.

The strings at the head of the boats from each cell are compared and the pointer corresponding to the lowest ranking string is put in the next free spot of an output boat. Pointer and string are removed from the input boat. When an output boat fills up, it is sent to the host; when an input boat empties, it is exchanged for the second boat from that cell and a new boat is requested.

A linear comparison of items at the head of each of the boats to be merged is currently made but it is planned to implement a selection tree as described by Knuth [5] to speed up the process. It is also expected that Batcher's method (also described in [5]) might produce faster overall sorting compared with the combined quicksort and ($n - 1$) way merge.

5 Stargazing At The Libraries Of The Future

Machines of the type of the AP1000 are equipped with what, not so long ago, seemed like astronomical amounts of high speed memory and have processing power to match. Further development of hardware technology make it likely that, within a few years, machines will be built with at least a terabyte (10^{12}X characters) of RAM. (A maximal configuration of the AP1000 presently has only 16 gigabytes of RAM, but production DRAMs with capacities sixteen times those employed in the AP1000 are likely to be available in the relatively near future. Further, it is not hard to envisage an increase by at least a factor of four in the maximum number of processing cells in an AP1000-like configuration.)

Equipped with high-performance text retrieval software, terabyte parallel machines could become the research libraries of the future. Assuming an average of 6 characters per word and one hundred thousand characters per volume, a terabyte of memory is sufficient to accommodate the entire textual data holdings of a library of one and a half million volumes.

There may be many practical, financial, legal, political and habitual reasons preventing the complete replacement of traditional libraries with parallel supercomputers. However, in a "library" based on an enormous-capacity, ultra-high performance, free-text retrieval engine , a number of advantages would accrue:

1. Several people can simultaneously read the same book,
2. No-one has to leave their desk (or home) to read a book from the library,
3. Books are never lost or incorrectly shelved,
4. Library opening hours can approach 168 hrs/week,
5. Author/title catalogues are never out of date,
6. Unlike current subject indexes which rely on someone else's keywords, a free-text retrieval system allows completely unfettered content addressability. The current problem of pockets of relevant but inaccessible information can be largely solved.

Free-text retrieval of the style embodied in PADDY, is eminently scalable provided that the text chunks on each cell do not become too short. Performance remains good provided the chunks are not too long relative to the processor speed. Let us imagine a hypothetical 4-terabyte AP1000-like machine, with 16K processors, each four times faster on memory intensive operations than the current AP1000 cells, and with 256 Mbyte of RAM per processor. Our one and a half million volume library, with its terabyte of data would occupy only 64 Mbytes of data in each cell, leaving generous provision for indexes and temporary search results.

Most library searches would cover only a restricted fraction of the collection, but, if necessary, the entire library collection could be scanned, character by character. On the basis of current results, BMG searches for six character strings over the entire terabyte of text are projected to take of the order of 6 seconds, a time not much longer than that of querying the on-line *catalogue* of some current libraries(!)

Even 6 seconds is too long for a multi-user interactive facility, however. Such a facility would need to be based on indexed searching, such as that described in [3]. Index-based searches over the entire text could be expected to take of the order of a tenth of a second.

It is estimated that building PADDY-style partial indexes would take about a quarter of an hour and that loading a terabyte of data from one thousand cell-connected, one gigabyte disks would take a similar time. A short period of downtime each night would thus permit complete index-rebuilding to incorporate new acquisitions. Backup of data could be provided with RAID techniques (and a thousand extra disks or so!).

Acknowledgements

Michael Hiron, working as a vacation student, coded the set and proximity operations and incorporated the regular expression and Boyer-Moore-Gosper code. His contribution is gratefully acknowledged. The co-operation of the Australian National Dictionary Centre has been vital to the project and work has also benefited from discussions with other members of the Department of Computer Science, particularly Dr P. Mackerras. The Boyer-Moore-Gosper and regular expression functions were supplied courtesy of the Free Software Foundation.

References

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

[2] Gonnet, G.H., Baeza-Yates, R.A. and Snider, T. *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.

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

[4] Hiron, M. and Hawking, D.A. *PADDY User Manual* Department of Computer Science, Australian National University, Canberra, Australia, Oct 1992.

[5] Knuth, D.M. *The Art Of Computer Programming* , vol. 3, "Sorting and Searching" Addison-Wesley, Reading, Massachusetts, 1973