BULK PROCESSING OF TEXT ON A MASSIVELY PARALLEL COMPUTER

Gary W. Sabot
Thinking Machines Corporation
245 First St.
Cambridge, MA 02142

Abstract

Dictionary lookup is a computational activity that can be greatly accelerated when performed on large amounts of text by a parallel computer such as the Connection Machine^{TM} Computer (CM). Several algorithms for parallel dictionary lookup are discussed, including one that allows the CM to lookup words at a rate 450 times that of lookup on a Symbolics 3600 Lisp Machine.

1 An Overview of the Dictionary Problem

This paper will discuss one of the text processing problems that was encountered during the implementation of the CM-Indexer, a natural language processing program that runs on the Connection Machine (CM). The problem is that of parallel dictionary lookup: given both a dictionary and a text consisting of many thousands of words, how can the appropriate definitions be distributed to the words in the text as rapidly as possible? A parallel dictionary lookup algorithm that makes efficient use of the CM hardware was discovered and is described in this paper.

It is clear that there are many natural language processing applications in which such a dictionary algorithm is necessary. Indexing and searching of databases consisting of unformatted natural language text is one such application. The proliferation of personal computers, the widespread use of electronic memos and electronic mail in large corporations, and the CD-ROM are all contributing to an explosion in the amount of useful unformatted text in computer readable form. Parallel computers and algorithms provide one way of dealing with this explosion.

2 The CM: Machine Description

The CM consists of a large number number of processor/memory cells. These cells are used to store data structures. In accordance with a stream of instructions that are broadcast from a single conventional *host* computer, the many processors can manipulate the data in the nodes of the data structure in parallel.

Each processor in the CM can have its own local variables. These variables are called *parallel variables*, or *parallel fields*. When a host computer program performs a serial operation on a parallel variable, that operation is performed separately in each processor in the CM. For example, a program might compare two parallel string variables. Each CM processor would execute the comparison on its own local data and produce its own local result. Thus, a single command can result in tens of thousands of simultaneous CM comparisons.

In addition to their computation ability, CM processors can communicate with each other via a special hardware communication network. In effect, communication is the parallel analog of the pointer-following executed by a serial computer as it traverses the links of a data structure or graph.

3 Dictionary Access

A dictionary may be defined as a mapping that takes a particular word and returns a group of *status bits*. Status bits indicate which sets or groups of words a particular word belongs to. Some of the sets that are useful in natural language processing include syntactic categories such as nouns, verbs, and prepositions. Programs also can use semantic characterization information. For example, knowing whether a word is name of a famous person (i.e. Lincoln, Churchill), a place, an interjection, or a time or calendar term will often be useful to a text processing program.

The task of looking up the definition of a word consists of returning a binary number that contains 1's only. in bit positions that correspond with the groups to which that word belongs. Thus, the definition of "Lincoln" contains a zero in the bit that indicates a word can serve as a verb, but it contains a 1 in the *famous person's name* bit.

While all of the examples in this paper involve only a few words, it should be understood that the CM is efficient and cost effective only when large amounts of

Figure 1. Simple Broadcasting Dictionary Algorithm, marking famous names

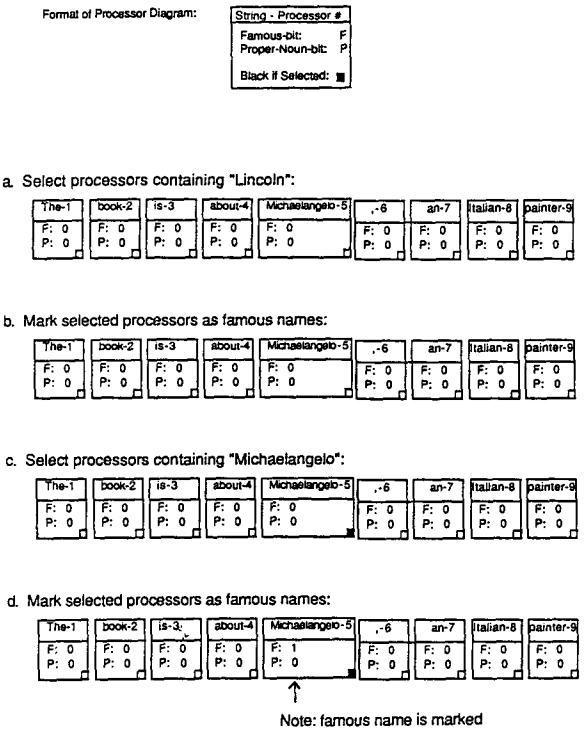
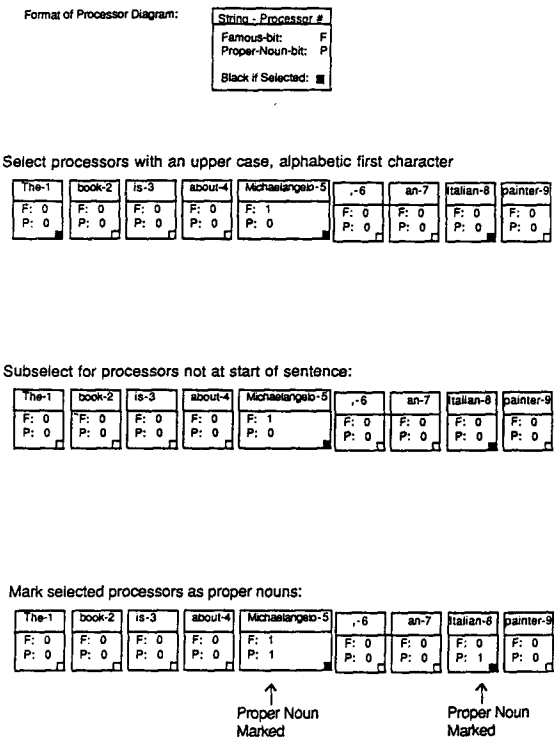


Figure 2. Syntactic Proper Noun Locator



text are to be processed. One would use the dictionary algorithms described in this paper to look up all of the words in an entire novel; one would not use them to look up the ten words in a user's query to a question answering system.

4 A Simple Broadcasting Dictionary Algorithm

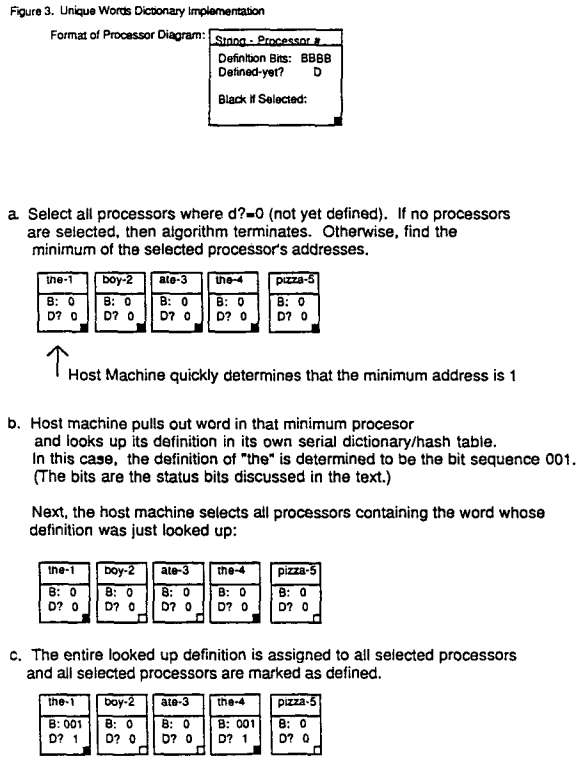
One way to implement a parallel dictionary is to serially broadcast all of the words in a given set. Processors that contain a broadcast word check off the appropriate status bits. When all of the words in one set have been broadcast, the next set is then broadcast. For example, suppose that the dictionary lookup program begins by attempting to mark the words that are also famous last names. Figure 1 illustrates the progress of the algorithm as the words "Lincoln" and then "Michaelangelo" are broadcast. In the first step, all occurrences of "Lincoln" are marked as famous names. Since that word does not occur in the sample sentence, no marking action takes place. In the second step, all occurrences of "Michaelangelo" are marked, including the one in the sample sentence.

In step d, where all processors containing "Michaelangelo" are marked as containing famous names, the program could simultaneously mark the selected processors as containing proper nouns. Such shortcuts will not be examined at this time.

After all of the words in the set of *famous names* have been broadcast, the algorithm would then begin to broadcast the next set, perhaps the set containing the names of the days of the week.

In addition to using this broadcast algorithm, the CM-Indexer uses syntactic definitions of some of the dictionary sets. For example, it defines a proper noun as a capitalized word that does not begin a sentence. (Proper nouns that begin a sentence are not found by this capitalization based rule; this can be corrected by a more sophisticated rule. The more sophisticated rule would mark the first word in a sentence as a proper noun if it could find another capitalized occurrence of the word in a nearby sentence.) Figure 2 illustrates the progress of this simple syntactic algorithm as it executes.

The implementation of both the broadcast algorithm and the syntactic proper noun rule takes a total of less than 30 lines of code in the *Lisp (pronounced "star-lisp") programming language. The entire syntactic rule that finds all proper nouns executes in less than 5 milliseconds. However, the algorithm that transmits word



lists takes an average of more than 5 milliseconds per word to broadcast a list of words from the host to the CM. Thus, since it takes time proportional to the number of words in a given set, the algorithm becomes a bottleneck for sets of more than a few thousand words. This means that the larger sets listed above (all nouns, all verbs, etc.) cannot be transmitted. The reason that this slow algorithm was used in the CM-Indexer was the ease with which it could be implemented and tested.

5 An Improved Broadcasting Dictionary Algorithm

One improvement to the simple broadcasting algorithm would be to broadcast entire definitions (i.e. several bits), rather than a single bit indicating membership in a set. This would mean that each word in the dictionary would only be broadcast once (i.e. "fly" is both a noun and a verb). A second improvement would be to broadcast only the words that are actually contained in the text being looked up. Thus, words that rarely occur in English, which make up a large percentage of the dictionary, would rarely be broadcast.

In summary, this improved dictionary broadcasting algorithm will loop for the unique words that are contained in the text to be indexed, look up the definition of each such word in a serial dictionary on the host machine, and broadcast the looked-up definition to the entire CM. Figure 3 illustrates how this algorithm would assign the definition of all occurrences of the word "the" in a sample text. (Again, in practice the algorithm operates on many thousands of words, not on one sentence.)

In order to select a currently undefined word to look up, the host machine executing this algorithm must determine the address of a selected processor. The figure indicates that one way to do this is to take the minimum address of the processors that are currently selected. This can be done in constant time on the CM.

This improved dictionary lookup method is useful when the dictionary is much larger than the number of unique words contained in the text to be indexed. However, since the same basic operation is used to broadcast definitions as in the first algorithm, it is clear that this second implementation of a dictionary will not be feasible when a text contains more than a few thousand unique words.

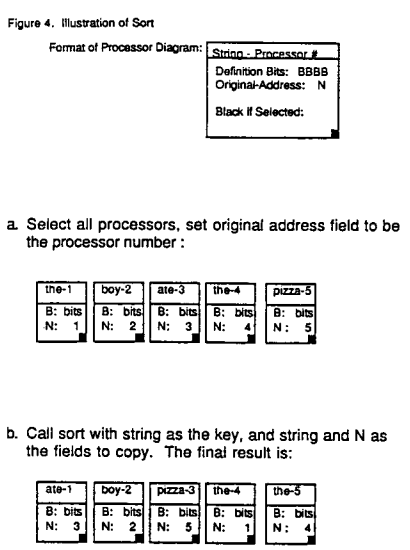
By analyzing a number of online texts ranging in size from 2,000 words to almost 60,000 words, it was found that as the size of the text approaches many tens of thousands of words, the number of unique words increased into the thousands. Therefore, it can be concluded that the second implementation of the broadcasting dictionary algorithm is not feasible when there are more than a few tens of thousands of words in the text file to be indexed.

6 Making Efficient Use of Parallel Hardware

In both of the above algorithms, the "heart" of the dictionary resided in the serial host. In the first case, the heart was the lists that represented sets of words; in the second case, the heart was the call to a serial dictionary lookup procedure. Perhaps if the heart of the dictionary could be stored in the CM, alongside the words from the text, the lookup process could be accelerated.

7 Implementation of Dictionary Lookup by Parallel Hashing

One possible approach to dictionary lookup would be to create a hash code for each word in each CM processor in parallel. The hash code represents the address of a different processor. Each processor can then send a lookup request to the processor at the hash-code address, where



the definition of the word that hashes to that address has been stored in advance. The processors that receive requests would then respond by sending back the prestored definition of their word to the address contained in the request packet.

One problem with this approach is that *all* of the processors containing a given word will send a request for a definition to the same hashed address. To some extent, this problem can be ameliorated by broadcasting a list of the n (i.e. 200) most common words in English, before attempting any dictionary lookup cycles. Another problem with this approach is that the hash code itself will cause collisions between different text words that hash to the same value.

8 An Efficient Dictionary Algorithm

There is a faster and more elegant approach to building a dictionary than the hashing scheme. This other approach has the additional advantage that it can be built from two generally useful submodules each of which has a regular, easily debugged structure.

The first submodule is the *sort* function, the second is the *scan* function. After describing the two submodules, a simple version of the fast dictionary algorithm will be presented, along with suggestions for dealing with memory and processor limitations.

8.1 Parallel Sorting

A parallel sort is similar in function to a serial sort. It accepts as arguments a parallel data field and a parallel comparison predicate, and it sorts among the selected processors so that the data in each successive (by address) processor increases monotonically. There are parallel sorting algorithms that execute in time proportional to the square of the logarithm of the number of items to be sorted. One easily implemented sort, the enumerate-and-pack sort, takes about 1.5 milliseconds per bit to sort 64,000 numbers on the CM. Thus, it takes 48 milliseconds to sort 64,000 32-bit numbers.

Figure 4 illustrates the effect a parallel sort has on a single sentence. Notice that pointers back to the original location of each word can be attached to words before the textual order of the words is scrambled by the sort.

8.2 Scan: Spreading Information in Logarithmic Time

A scan algorithm takes an associative function of two arguments, call it F, and quickly applies it to data field values in successive processors of:

- a
- b
- c
- d
- e

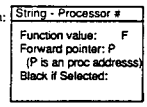
The scan algorithm produces output fields in the same processors with the values:

- a
- F(a, b)
- F(F(a, b), c)
- F(F(F(a, b), c), d)
- etc.

The key point is that a scan algorithm can take advantage of the associative law and perform this task in logarithmic time. Thus, 16 applications of F are sufficient to scan F across 64,000 processors. Figure 5 shows one possible scheme for implementing scan. While the scheme in the diagram is based on a simple linked list structure, scan may also be implemented on binary trees, hypercubes, and other graph data structures. The nature of the routing system of a particular parallel computer will select which data structures can be scanned most rapidly and efficiently.

Figure 5. Illustration of Scan

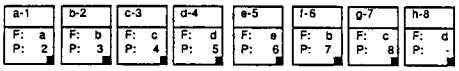
Format of Processor Diagram:



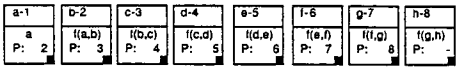
Backward pointer can be calculated in constant time: all processors send their own addresses to the processors pointed to by P.

f is any associative function of two arguments

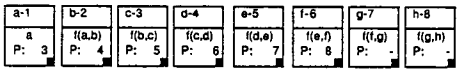
a. Select all processors, initialize function value to string, forward pointer to self address + 1:



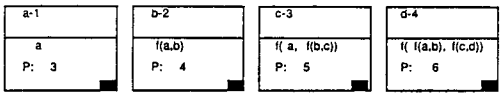
b. Get back pointer, get function value from processor at back pointer, call this value BF. Replace the current function value, F, with f(BF,F):



c. Calculate a forward pointer that goes twice as far as the current forward pointer. This can be done as follows: Get the value of P at the processor pointed to by your own P, and replace your own P with that new value:



d. If any processor has a valid forward pointer, goto b (the next execution of b has the following effect on the first 4 processors:



(Note that since f is associative, f(a, f(b, c)) is always equal to f(f(a,b), c), and f(f(a,b), f(c,d)) = f(f(f(a, b), c), d)

When combined with an appropriate F, scan has applications in a variety of contexts. For example, scan is useful in the parallel enumeration of objects and for region labeling. Just as the FFT can be used to efficiently solve many problems involving polynomials, scan can be used to create efficient programs that operate on graphs, and in particular on linked lists that contain natural language text.

8.3 Application of Scan and Sort to Dictionary Lookup

To combine these two modules into a dictionary, we need to allocate a bit, DEFINED?, that is 1 only in processors that contain a valid definition of their word. Initially, it is 1 in the processors that contain words from the dictionary, and 0 in processors that contain words that come from the text to be looked up. The DEFINED? bit will be used by the algorithm as it assigns definitions to text words. As soon as a word receives its definition, it will have its DEFINED? bit turned on. The word can then begin to serve as an additional copy of the dictionary entry for the remainder of the lookup cycle. (This is the "trick" that allows scan to execute in logarithmic time.)

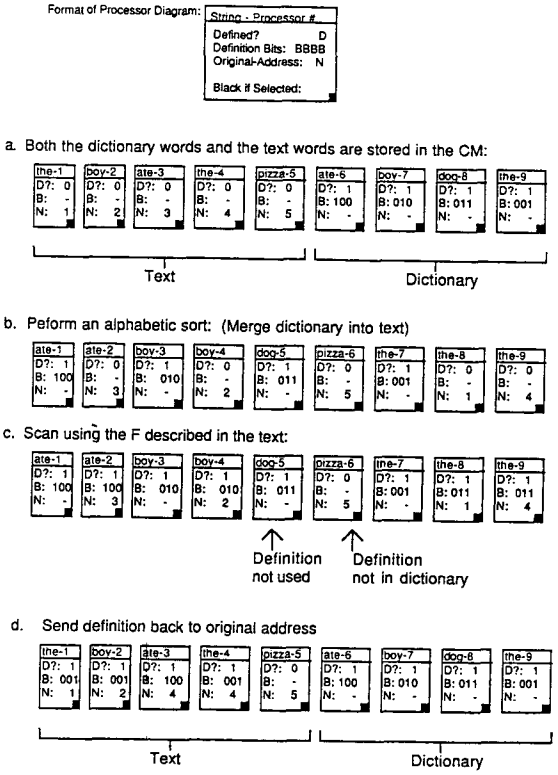
First, an alphabetic sort is applied in parallel to all processors, with the word stored in each processor serving as the primary key, and the DEFINED? bit acting as a secondary key. The result will be that all copies of a given word are grouped together into sequential (by processor address) lists, with the single dictionary copy of each word immediately preceding any and all text copies of the same word.

The definitions that are contained in the dictionary processors can then be distributed to all of the text words in logarithmic time by scanning the processors with the following associative function f:

```
;; x and y are processors that have the following
;; fields or parallel variables:
;;  STRING  (a word)
;;  DEFINED?  (1 if word contains a correct definition)
;;  ORIGINAL-ADDRESS (where word resided before sort)
;;  DEFINITION (initially correct only in dictionary
;;              words)

;; function f returns a variable containing the same
;; four fields.  This is a pseudo language; the actual
;; program was written in *Lisp.

function f(x,y):
   f.STRING = y.STRING
   f.ORIGINAL-ADDRESS = y.ORIGINAL-ADDRESS
   if y.DEFINED? = 1 then
           {
           ;; if y is defined, just return y
           f.DEFINED? = 1
           f.DEFINITION = y.DEFINITION
           }
   else
           if x.STRING = y.STRING then
                   {
                   ;; if words are the same, take
                   ;; any definition that x may have
                   f.DEFINED? = x.DEFINED?
                   f.DEFINITION = x.DEFINITION
                   }
           else
                   ;; no definition yet
                   f.DEFINED? = 0

;; note that text words that are not found in the
;; dictionary correctly end up with DEFINED? = 0
```

This function F will spread dictionary definitions from a definition to all of the words following it (in processor address order), up until the next dictionary word. Therefore, each word will have its own copy of the dictionary definition of that word. All that remains is to have a single routing cycle that sends each definition back to the original location of its text word. Figure 6 illustrates the execution of the entire sort-scan algorithm on a sample sentence.

Figure 6. Illustration of Sort-Scan Algorithm



8.4 Improvements to the Sort-Scan Dictionary Algorithm

Since the CM is a bit serial Machine, string operations are relatively expensive operation. The dictionary function F described above performs a string comparison and a string copy operation each time it is invoked. On a full size CM, the function is invoked 16 times (log 64K words). A simple optimization can be made to the sort-scan algorithm that allows the string comparison to be performed only once. This allows a faster dictionary function that performs no string comparisons to be used.

The optimization consists of two parts. First, a new stage is inserted after the sorting step, before the scanning step. In this new step, each word is compared to the word to its left, and if it is different, it is marked as a "header." Such words begin a new segment of identical words. All dictionary words are headers, because the sort places them before all occurrences of identical words. In addition, the first word of each group of words that does not occur in the dictionary is also marked as a header.

Next, the following function creates the field that will be scanned:

```
;; header-p is a parallel boolean variable that is
;; true in headers, false otherwise

function create-field-for-scan(header-p):

;;define a type for a large bit field
   var FIELD : record

          ;;most significant bits contain
          ;;processor address
          ADDRESS

          ;;least significant bits will
          ;;contain the definition
          DEFINITION
        end

;;initialize to address 0, no definition
  FIELD.ADDRESS = 0
  FIELD.DEFINITION = 0

;; next, the headers that are dictionary words store
;; their definitions in the correct part of FIELD
;;  Non-dictionary headers (text words not found
;;  in dictionary) are given null definitions.

   if header
        {
        FIELD.DEFINITION = definition

        ;; self-address contains each processor\'s
        ;; own unique address
        FIELD.ADDRESS = self-address
        }
   return(FIELD)
```

Finally, instead of scanning the dictionary function across this field, the maximum function (which returns the maximum of two input numbers) is scanned across it. Definitions will propagate from a header to all of the words within its segment, but they will not cross past the next header. This is because the next header has a greater self-address in the most significant bits of the field being scanned, and the maximum function selects it rather than the earlier headers smaller field value. If a header had no definition, because a word was not found in the dictionary, the null definition would be propagated to all copies of that word.

The process of scanning the maximum function across a field was determined to be generally useful. As a result, the max-scan function was implemented in an efficient pipelined, bit-serial manner by Guy Blelloch, and was incorporated into the general library of CM functions.

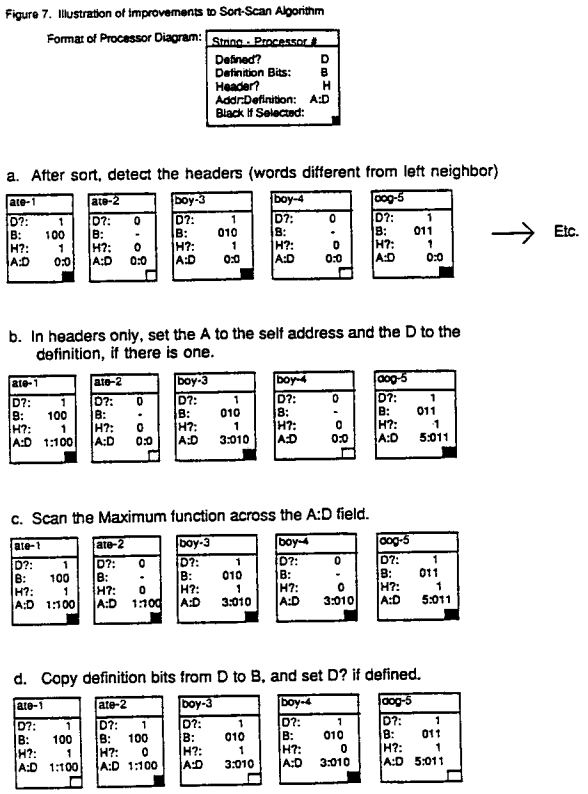


Figure 7 illustrates the creation of this field, and the scanning of the maximum function across it. Note that the size of the field being scanned is the size of the definition (8 bits for the timings below) plus the size of a processor address (16 bits). In comparison, the earlier dictionary function had to be scanned across the definition and the original address, along with the entire string. Scanning this much larger field, even if the dictionary function was as fast as the maximum function, would necessarily result in slower execution times.

8.5 Evaluation of the Sort-Scan Dictionary Algorithm

The improved sort-scan dictionary algorithm is much more efficient than the broadcasting algorithms described earlier. The algorithm was implemented and timed on a Connection Machine.

In a bit-serial computer like the CM, the time needed to process a string grows linearly with the number of bits used to store the string. A string length of 8 characters is adequete for the CM-Indexer. Words longer than 8 characters are represented by the simple concatenation of their first 4 and last 4 characters. ASCII characters therefore require 64 bits per word in the CM; 4 more bits are used for a length count.

Because dictionary lookup is only performed on alphabetic characters, the 64 bits of ASCII data described above can be compacted without collision. Each of the twenty-six letters of the alphabet can be represented using 5 bits, instead of 8, thereby reducing the length of the character field to 40 bits; 4 bits are still needed for the length count. Additional compression could be achieved, perhaps by hashing, although that would introduce the possibility of collisions. No additional compression is performed in the prototype implementation. The timings given below assume that each processor stores an 8 character word using 44 bits.

First of all, to sort a bit field in the CM currently takes about 1.5 milliseconds per bit. Second, the function that finds the header words was timed and took less than 4 milliseconds to execute. The scan of the max function across all of the processors completed in under in 2 milliseconds. The routing cycle to return the definitions to the original processors of the text took approximately one millisecond to complete.

As a result, with the improved sort-scan algorithm, an entire machine full of 64,000 words can be looked up in about 73 milliseconds. In comparison to this, the original sort-scan implementation requires an additional 32 milliseconds (2 milliseconds per invocation of the slow dictionary function), along with a few more milliseconds for the inefficient communications pattern it requires.

This lookup rate is approximately equivalent to a serial dictionary lookup of .9 words per microsecond. In comparison, a Symbolics Lisp Machine can look up words at a rate of 1/500 words per microsecond. (The timing was made for a lookup of a single bit of information about a word in a hash table containing 1500 words). Thus, the CM can perform dictionary lookup about 450 times faster than the Lisp Machine.

8.6 Coping with Limited Processor Resources

Since there are obviously more than 64,000 words in the English language, a dictionary containing many words will have to be handled in sections. Each dictionary processor will have to hold several dictionary words, and the look-up cycle will have to be repeated several times. These adjustments will slow the CM down by a multiplicative factor, but Lisp Machines also slow down when large hash tables (often paged out to disk) are queried.

There is an alternative way to view the above algorithm modifications: since they are motivated by limited processor resources, they should be handled by some sort of run time package, just as virtual memory is used to handle the problem of limited physical memory resources on serial machines. In fact, a virtual processor facility is currently being used on the CM.

9 Further Applications of Scan to Bulk Processing of Text

The scan algorithm has many other applications in text processing. For example, it can be used to lexically parse text in the form of 1 character per processor into the form of 1 word per processor. Syntactic rules could rapidly determine which characters begin and end words. Scan could then be used to enumerate how many words there are, and what position each character occupies within its word. The processors could then use this information to send their characters to the word-processor at which they belong. Each word-processor would receive the characters making up its word and would assemble them into a string.

Another application of scan, suggested by Guy L. Steele, Jr., would be as a regular expression parser, or lexer. Each word in the CM is viewed as a transition matrix from one set of finite automata states to another set. Scan is used, along with an F which would have the effect of composing transition matrices, to apply a finite automata to many sentences in parallel. After this application of scan, the last word in each sentence contains the state that a finite automata parsing the string would reach. The lexer's state transition function F would be associative, since string concatenation is associative, and the purpose of a lexer is to discover which particular strings/tokens were concatenated to create a given string/file.

The experience of actually implementing parallel natural language programs on real hardware has clarified which operations and programming techniques are the most efficient and useful. Programs that build upon general algorithms such as sort and scan are far easier to debug than programs that attempt a direct assault on a problem (i.e. the hashing scheme discussed earlier; or a slow, hand-coded regular expression parser that I implemented). Despite their ease of implementation, programs based upon generally useful submodules often are more efficient than specialized, hand-coded programs.

Acknowledgements

I would like to thank Dr. David Waltz for his help in this research and in reviewing a draft of this paper. I would also like to thank Dr. Stephen Omohundro, Cliff Lasser, and Guy Blelloch for their suggestions concerning the implementation of the dictionary algorithm.

References

Akl, Selim G. *Parallel Sorting Algorithms*, 1985, Academic Press, Inc.

Feynman, Carl Richard, and Guy L. Steele Jr. *Connection Machine Macroinstruction Set, REL 2.3.*, Thinking Machines Corporation. (to appear)

Hillis, W. Daniel. *The Connection Machine*, 1985, The MIT Press, Cambridge, MA.

Lasser, Clifford A., and Stephen M. Omohundro. *The Essential *Lisp Manual*, Thinking Machines Corporation. (to appear)

Leiserson, Charles, and Bruce Maggs. "Communication-Efficient Parallel Graph Algorithms," Laboratory for Computer Science, Massachusetts Institute of Technology. (to appear) (Note: scan is a special case of the *treefix* algorithm described in this paper.)

Omohundro, Steven M. "A Connection Machine Algorithms Primer," Thinking Machines Corporation. (to appear)

Resnikoff, Howard. *The Illusion of Reality*, 1985, in preparation.

Waltz, David L. and Jordan B. Pollack. "Massively Parallel Parsing: A Strongly Interactive Model of Natural Language Interpretation," *Cognitive Science*, Volume 9, Number 1, pp. 51-74, January-March, 1985.