Methods of Large Grammar Representation in Massively Parallel Parsing Systems*

Stefan Winz and James Geller
CIS Department
New Jersey Institute of Technology
Newark, NJ 07102
geller@vienna.njit.edu
January 13, 1993

Abstract

This paper describes techniques for massively parallel parsing where sequences of lexical categories are assigned to single processors and compared in parallel to a given input string. Because even small grammars result in full expansions that are much larger than the largest existing massively parallel computers, we need to develop techniques for "doubling up" sequences on processors so that they don't interfere during parallel matching. This paper describes three such techniques: (1) discrimination by length, (2) discrimination by open class/closed class words, and (3) combined discrimination by length and word class. We discuss possible reductions of the sequence space and implementation techniques on a CM-5 Connection Machine**.

*This work was performed using the computational resources of the Northeast Parallel Architecture Center at Syracuse University, which is funded by and operates under contract to DARPA and the Airforce Systems Command, Rome Air Development Center, Griffiss Airforce Base, NY, under contract #F306002-88-C-0031.

**Connection Machine is a trademark of Thinking Machines Inc.

I. INTRODUCTION

Various methods of utilizing the computational power of parallel processing in natural language understanding have been discussed in the past. [1] outlined a method to parallelize natural language processing by using a dataflow approach to parsing. Work on parallel speech recognition has been reported in [14]. Massively parallel parsing from a connectionist point of view has also been reported, e.g. in [11]. This paper examines parallel natural language processing from a slightly different angle.

A new approach to parsing similar to the one of Kitano and Higuchi [4,5,6,7], which they refer to as "Massively Parallel Memory-Based Parsing" is discussed here. This method places a great deal of importance on the use of memory and parallel processing. These elements replace the lengthy serial operations performed by conventional parsing methods. The fastest of these conventional methods, Tomita's parser, still only attains a parse within $O(n^3)$ time [15].

Upon developing such a system we noticed that even for relatively simple context free grammars the number of processing elements necessary is much larger than currently existing massively parallel machines. Kitano's [4,8] approach has been to limit sentence patterns to those observed in real text. Our approach described in this paper is to develop techniques for representing larger grammars with fewer processing elements without losing the speed or elegance of the original parallel algorithm.

The remainder of this Section and Section II of this paper discusses the basic ideas behind massively parallel memory based parsing. In Sections III, IV and V, three solutions to our problem are discussed in detail. Algorithms and analyses are presented there. Section VI discusses implementations and presents conclusions. This paper assumes that the reader has a basic grasp of parallel processing and architectures such as the Connection Machine [3] or the IXM2 [2].

MASSIVELY PARALLEL MEMORY-BASED PARSING

The basic idea of massively parallel memory based parsing [4] is to eliminate the serial rule application and search of conventional parsing methods. Memory based parsing is based on memory based reasoning [10,12,13], which places memory at the heart of intelligence. Parsing is performed by a parallel search of memory. Input sentences are compared with past occurrences of sentences that are stored in memory. The interpretation of the input sentence is based upon recalling the interpretation of a past occurrence.

Keeping a large enough corpus of sentences to parse a realistic number of input sentences, even from a limited domain, would require a tremendous amount of memory. This is true even if the actual implementation stores only sentence abstractions instead of actual sentences. For example, the sentence "The man walked the dog" would be represented in memory as: [art noun verb art noun]. This would also represent all sentences similar to the above. The drawback is that semantically incorrect sentences such as: "The flower smiled the cat" would also be represented. However, further semantic processing would flag this type of sentence as an error.

There are two means by which one could generate the syntactic structures stored in memory. The first is to take a predefined grammar and expand it until all structures contain only terminal symbols such as 'art', 'noun', etc. The second means is to process a corpus of sentences. The result of this processing would be a set of syntactic structures. The latter is the approach taken by Kitano and Higuchi [8]. A description of the former, which we call the "fully expanded grammar" approach, follows.

II. THE FULLY EXPANDED GRAMMAR APPROACH

The first task that the parsing system must perform is a preprocessing step, it fully expands the grammar resulting in a set of all possible syntactic structures allowed by that grammar. These syntactic structures are sequences containing only terminal symbols. To avoid infinite sequences that would be the result of recursive rules, the grammar has to be "unrolled" by allowing rules with repeated terminal symbols and prohibiting recursive rules (e.g.: the rewrite rules NP ← ADJ noun, ADJ ← adj ADJ, ADJ ← adj, get "unrolled" to NP ← adj noun, NP ← adj adj noun). It should be an acceptable solution to the problem because most sentence do not go too "deep" into the recursion. Very infrequently is a sentence encountered which has more than four adjectives which modify the noun, for instance. Therefore, if the recursive rule above were "unrolled" so that it encompasses up to four adj modifiers for one noun, then most feasible sentence cases would be covered.

The next stage entails the transfer of one syntactic structure to each processor. The syntactic structure is stored as a group of parallel variables. In addition, each processor will have the length of the structure it is storing represented as a parallel variable. For example, if processor #2 is storing the structure: [art noun aux verb], then processor #2's parallel variable "length" would contain 4, the number of lexical elements in the structure. After the preprocessing stage is finished, the parsing process can be broken down into two phases. Phase one is to transform the input sentence into a specific format. The second phase takes the output of phase one and performs a parallel comparison operation with the parallel variables representing all possible syntactic structures.

Phase one is executed on the front-end. It first reads in the sentence to be parsed. It then performs a lexical lookup on all the words in the sentence. If a word appears that is not in the lexicon, it immediately fails. Each word is replaced by all the syntactic categories it fits into.

For example, if the sentence "Airplanes are landing" is read in and the front-end finds the following entries in the lexicon:

Airplanes → noun
are → aux, verb
landing → verb, adj

then the literal words are replaced by: {(noun) (aux verb) (verb adj)}. Phase two consists of two steps, both of which are parallel operations whose purpose is to "deactivate" processors. After these operations are completed, only processing elements which contain matching syntactic structures will be left active.

Initially all processors are active. In step one of phase two the algorithm is deactivating all processors whose syntactic structure is not equal in length to that of the input sentence. Obviously, such structures could never constitute a match. The second step builds a string of parallel AND operations consisting of parallel OR operations that perform all necessary comparisons of the whole input sequence with all stored sequences in parallel. The first position of the syntactic structures must match **one** of the categories of the first word of the input sentence **AND** the second position of the syntactic structures must match **one** of the categories of the second word of the input sentence **AND** ..... etc. All processors for which this is not the case are deactivated. Any processor left active after this operation represents a possible "parse" of the input sentence. This matching is done in parallel and is thus a very fast operation. A detailed example best illustrates this procedure.

Consider the following processors and their values:

Processor #1: [noun aux adj noun]
length = 4

Processor #2: [noun verb noun]
length = 3

Processor #3: [name verb noun]
length = 3

Processor #4: [noun aux verb]
length = 3

Using the same input sentence as above ("Airplanes are landing"), the first step would deactivate Processor #1 because the length of its structure is not equal to the input length. The second step would deactivate both Processor #2 and Processor #3. This is due to the fact that structure position one must match "noun" (eliminating Processor #3) AND position two must match "aux" OR "verb" (eliminating nothing) AND position 3 must match either "verb" OR "adj" (eliminating Processor #3). This leaves Processor #4 as the only active processor and its syntactic structure as the only possible "parse" of the input according to our grammar.

It is possible that there is more than one valid syntactic structure for a single sentence. Consider the following sentences:

(1) "We were hiding drugs"
(2) "We were hiding fugitives"

After parsing, both could be represented by the following valid syntactic structures:

(a) [pronoun verb adj noun]
(b) [pronoun aux verb noun]

It is obvious to us that sentence (1) cannot possibly have the structure of (a), because in order for "hiding" to be an "adj", the object which it is modifying must be animate. The parser has no way of knowing this and passes both on as valid structures. The determination of which structure is correct has to be made by semantic processing. Sentence (2), however, can have semantic meaning with both structures (a) and (b). Humans can be fugitives which are hiding or can hide fugitives. Further semantic processing will not result in a determination of which is correct. Contextual interpretation may or may not result in a determination. A sentence such as (2) is the reason that the parser allows multiple valid parses of a single sentence.

FULLY EXPANDED GRAMMAR APPROACH PROBLEMS

Consider the grammar given below. [**Figure 1**]

1) S ← NP VP
2) S ← NP VP NP
3) S ← NP VP PP
4) S ← NP VP NP NP
5) S ← NP PP VP

6) NP ← name
7) NP ← noun
8) NP ← art noun
9) NP ← art adj noun
10) NP ← adj noun
11) NP ← art adj adj noun
12) NP ← adj adj noun

13) VP ← verb
14) VP ← aux verb
15) VP ← modal verb
16) VP ← modal aux verb
17) VP ← aux aux verb
18) VP ← modal aux aux verb
{aux != aux is assumed}

19) PP ← prep NP
20) PP ← prep NP prep NP

The expansion of this simple grammar results in 7,098 syntactic structures. Before using this grammar, we experimented with another slightly more complex grammar, which resulted in 6,743,226 expansions. Furthermore, if we were to use an even more complex grammar such as one which includes embedded sentences, relative clauses, interrogative and imperative statements, particles and conjunctions then we can expect an astronomical number of expansions. Typical massively parallel computers have between 1k and 64k of processors.

Obviously we cannot assign only one expansion per processor. Sections III, IV and V of this paper examine three possible solutions to this problem.

III. APPROACH ONE: USING LENGTH DISCRIMINATION

The first approach is based on the fact that sentences come in different lengths. At least as far as our example grammar is concerned, there seems to be a good distribution of syntactic structure length. By placing syntactic structures of different lengths on one processor, more syntactic structures can be represented without increasing the number of processors or compromising the speed, elegance or simplicity of the fully expanded grammar approach.

Before describing in more detail the workings of such a system, let's examine the distribution of syntactic structure lengths for different grammars. We performed an exhaustive analysis of five progressively more complicated grammars. The grammar previously mentioned will generate 7,098 syntactic structures, of which 2 expansions are of length two, 10 are of length three, 42 are of length four, etc. The longest expansion, of which there are 2 occurences, is of length eighteen. Of all the expansions, there are more of length ten than of any other single length. To be exact, there are 1,118 expansions of length ten, which is roughly 16% of the original 7,098.

The most complex grammar that we analyzed results in 87,654 expansions. In this grammar, there are more expansions of length thirteen than of any other length. More precisely, 13,514 expansions are of length thirteen. The expansion and tabulation of this grammar took approximately 28 hours on a DECsystem 5900 using a MIPS R3000 processor running Kyoto Common Lisp. Furthermore, several intermediate grammars resulting in between 7,098 and 87,654 expansions exhibited similar trends. That is, the most common length subcategory made up approximately 15% of the total number of expansions.

The preprocessing stage begins by a grammar expanding into syntactic structures. Upon completion, one structure of every length is placed on the k-th processor. Once a sentence to be parsed is input, all structures on all processors not of the input length will be ignored. Thus only one structure on every processor will be both active and potentially valid. The algorithm to be applied once the preprocessing stage has terminated and the structures have been loaded onto the processors is as follows:

Algorithm: Discrimination by Length

Step One -

(1) Read the input sentence from the user.

(2) Look up each word of the input sentence in the lexicon. With this information, construct every possible syntactic structure which represents that sentence.

Step Two -

(3) Activate all processors.

(4) Identify those parallel variables that contain structures whose length is equal to the length of the input sentence.

(5) Deactivate all the processors on which the parallel variables identified in step (4) do not match one of the input structures in (2).

(6) Any processor remaining active represents an acceptable "parse" of the input sentence.

Step One is performed entirely on the front end computer while Step Two is performed in parallel on the processing elements. In the implementation, (5) is executed using a macro which expands into a group of parallel OR statements inside a parallel AND statement. The actual sequence is stored as a parallel array. The OR statements match each position in the array against each possible syntactic category at the corresponding word position of the input sentence. This AND-OR combination is then executed in parallel. To see that this algorithm operates correctly, refer back to the example in Section II.

Discrimination by length has the following advantages:

- The process has retained the elegance and simplicity of the fully expanded grammar approach.
- It utilizes the processor's local memory efficiently without introducing any serial components into the algorithm.
- Assuming enough local processor memory, a fully expanded grammar resulting in X syntactic structures can be handled by only about 0.15 * X processors.

The main disadvantages are:

• The reduction of processor requirements may still not be sufficient to handle extensive grammars.
• Local processor memory is used for additional syntactic structures which might not leave enough space for other necessary information, such as parse trees for the recognized syntactic structures.

IV. APPROACH TWO: DISCRIMINATION BY CLOSED CLASS WORDS

Lexical categories themselves are separated into two groups: those that belong to a closed class and those that belong to an open class [9]. A closed class is one which has a fixed number of words and the invention of new members is not to be expected. For example, the 'article' category is closed because there is a fixed group of articles in the English language and no more will be added. Other closed classes would be pronouns, conjunctions, prepositions, auxiliaries, modals, etc. An open class is one which is constantly growing. Nouns, verbs, adverbs and adjectives are open classes.

In this approach we "collapse" a large set of syntactic structures into a much smaller set of "open-closed" sequences. In this way we reduce the number of necessary processors because one open-closed sequence represents several different syntactic structures.

Our previously listed grammar in Fig. 1 produces 7,098 syntactic structures. This collapses into 1,850 open-closed sequences, which is 26% of the original amount. As already noted, this reduction is due to several different syntactic structures having identical open-closed sequences. The one open-closed sequence which occurs most frequently represents 24 different syntactic structures. For example, both [noun modal verb] and [noun aux verb] are represented by <open closed open>.

Our largest grammar, which results in 87,654 syntactic structures, collapses into 12,417 open-closed sequences. The "unique" open-closed sequences amount to roughly 14% of the number of syntactic structures. The open-closed sequence which represents the most expansions represents 86 expansions for this grammar.

Furthermore, other tests performed on grammars resulting in between 7,098 to 87,654 structures show that the reduction percentage declines steadily as the number of syntactic structures grows. In the example with 7,098 syntactic structures the reduction percentage was 26%. This percentage declines to 14% for 87,654 syntactic structures. This decreasing tendency is due to the apparent trend that when the amount of syntactic structures increases, they collapse into "already established" open-closed sequences rather than defining new ones. At what percentage this "bottoms out" is a subject of future investigation. Figure 2 at the end of this paper graphically depicts this tendency.

The first step of preprocessing is to translate syntactic structures into sequences of O's and C's and then place one sequence on each processor. For example, the structure (noun verb) is translated into (O O) and this is then placed on a processor. In addition, each processor has two numeric parallel variables, Begin and End, a "pointer" variable, and a group of syntactic structure variables (each of which is an array containing a syntactic structure) with names such as SS1, SS2, ...SSn.

On any given processor, there will be one open-closed sequence. The syntactic structures which "collapse" into this open-closed sequence are stored on contiguous processors numbered Begin through End in the parallel variable pointed to by the pointer variable. In other words, the pointer variable will be set to "point" to SS1 or SS2, etc.

Algorithm: Discrimination by closed class words

(1) Read the input sentence from the user.

(2) Look up each word of the input sentence in the lexicon. With this information, construct every possible open-closed sequence which represents that sentence. Also, maintain every possible syntactic structure which could represent the input sentence (This will be used later and is a byproduct of this step).

(3) Activate all processors.

(4) Deactivate all processors on which the open-closed sequence parallel variables do not match one of the possible input sentence open-closed sequences constructed in (2).

(5) If no processors are left active, the parse fails. Otherwise one or several processors will be left active. For every active processor (referred to by the name "Pnum") do:

- Identify the syntactic structure parallel variable indicated by the pointer variable of processor Pnum and activate all processors between processor Pnum's Begin value and its End value.
- Deactivate all processors on which the parallel variable identified above does not match one of the

input sentence's possible syntactic structures generated in step (2)

- If any processors remain active, they represent acceptable syntactic structures for the input sentence.

Steps (1) and (2) are performed on the front-end. Steps (3), (4) and (5) are parallel operations performed on the processing elements. If (4) results in multiple active processors, the system loops through the possibilities as indicated. Because there are very few words that can belong to open and closed classes (such as "can"), such loops will usually run only once or a few times.

Example Run

To illustrate the algorithm, we will present an example. Assume we have performed all preprocessing steps already.

(1) We read in the sentence "The market crashed."

(2) In the lexicon, we find the following:

market	noun, verb
crashed	verb
the	art

Furthermore, we know that articles belong in the closed class and verbs and nouns belong in the open class. The possible syntactic structures for this sentence are:

{a} [art verb verb]
{b} [art noun verb]

In addition, every possible open-closed sequence for the input is:

{A} <C O O>

(3) Activate all processors.

(4) Deactivate all processors whose open-closed sequence variable does not match one of the sequences constructed in (2). In other words, deactivate it unless its open-closed sequence matches: {A}.

(5) Let us assume that only processor #56 is active after this. Furthermore, let us assume that it stores the following values:

Processor #56.

variable	value
open-closed seq.	<C O O>
Begin	21
End	22
Pointer	SS3
SS1	irrelevant here
SS2	irrelevant here
SS3	irrelevant here

All processors except those between Begin (with a value of 21) and End (with a value of 22) are now deactivated. Processor #56's pointer indicates SS3, so this is the syntactic structure variable to be operated upon. Processors 21 and 22 store the following values in the parallel variable SS3:

Processor	Value
21	[pro verb noun]
22	[art noun verb]

These sequences are then matched against the syntactic structures generated for the input sentence in step (2). In other words, in order to remain active, the SS3 variable of a processor must match either {a} OR

{b}. After this match is executed, only processor 22 is active because its SS3 variable matches {b}. It represents a valid syntactic structure for the input sentence.

The open-closed approach has the following advantages:

- It uses the processors more efficiently than the fully expanded grammar approach without any discrimination by representing more syntactic structures on fewer processors.
- The ratio between open-closed sequences and the syntactic structures they represent falls as the number of expansions produced by a grammar increases. If local memory is sufficiently available, this approach might represent the same number of syntactic structures on fewer processors than the length method.

Disadvantages are:

- This approach requires a serial step if there is more than one valid open-closed sequence for one input.
- The approach has lost some of the simplicity and elegance in comparison with the length approach.
- Even more local processor memory is needed.

V. Approach Three: Combined Discrimination

The last approach that will briefly be considered is the combination of length discrimination and word class discrimination. The application of both discrimination methods results in an approximate 2% reduction rate. Figure 3 at the end of the paper graphically depicts the "Reduction Curve" for all three methods presented so far. The bottom curve shows the reduction tendency for the Combined Approach. The algorithm will be set up similarly to the algorithm for the discrimination by closed classes. The preprocessing phase begins by separating the syntactic structures into length subcategories. They are then "collapsed" into open-closed sequences, resulting in groups of open-closed sequences segregated by length. There will be parallel variables for open-closed sequences for each length subcategory. For instance, if we have lengths ranging from 2 to 22, we would have parallel variables with the names OC2, OC3, OC4, ...OC22.

One open-closed sequence of every length will be placed on each processor in the above mentioned variables. Additionally, parallel variables named Begin, End, and Pointer serve the same purposes as their counterparts in the previous algorithm. However, instead of having just one of each of these variables, one of these variables will exist **for each** OCx variable mentioned above (i.e. Begin2, Begin3, Begin4, ... Begin22). Finally, the syntactic structure parallel variables will be either one or two dimensional arrays. A one dimensional array holds one syntactic structure, while a two dimensional array holds several (this can be envisioned as a table with several rows). Each processor will have one syntactic structure variable **of each length**. The value of the pointer parallel variables is used as an index into the syntactic structure arrays. For instance, a pointer value of 2 "points" to the second row of a syntactic structure array. It is also important to note that two structures of the same length on the same processor **must** have different open-closed sequences. The algorithm first attains the valid open-closed sequences. It then determines the proper syntactic structure parallel variables to consider and executes a parallel match to derive the acceptable syntactic structures for the input sentence. A more detailed description and the algorithm of the combined approach can be found in [16,17].

VI. Conclusion and Implementation

In this paper, we have examined problems with massively parallel parsing due to the mismatch between the number of processors of existing architectures and the number of expansions of even small non-recursive grammars. We introduced three techniques that allow the placement of several expansions on one processor without interfering with parallel matching: (1) Discrimination by length; (2) Discrimination by open class/closed class words; and (3) Combined Discrimination. Discrimination by length shows smaller reduction rates than discrimination by word class, but discrimination by word class introduces serial steps into the processing. The combined approach shows the best reduction percentage but it also requires some serial processing. All algorithms described in this paper have been implemented. Discrimination by length, discrimination by word class and the combined discrimination methods were implemented on a CM-5 Connection Machine using 16k of virtual processors. The programming language used was *LISP.

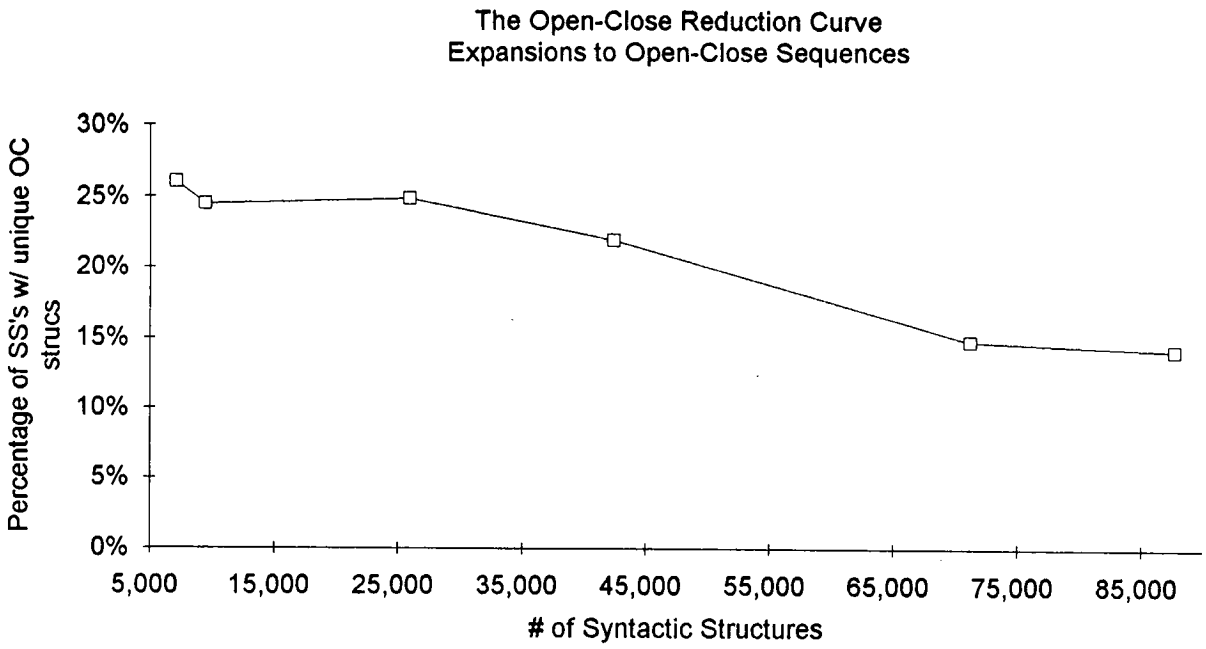

Figure 2

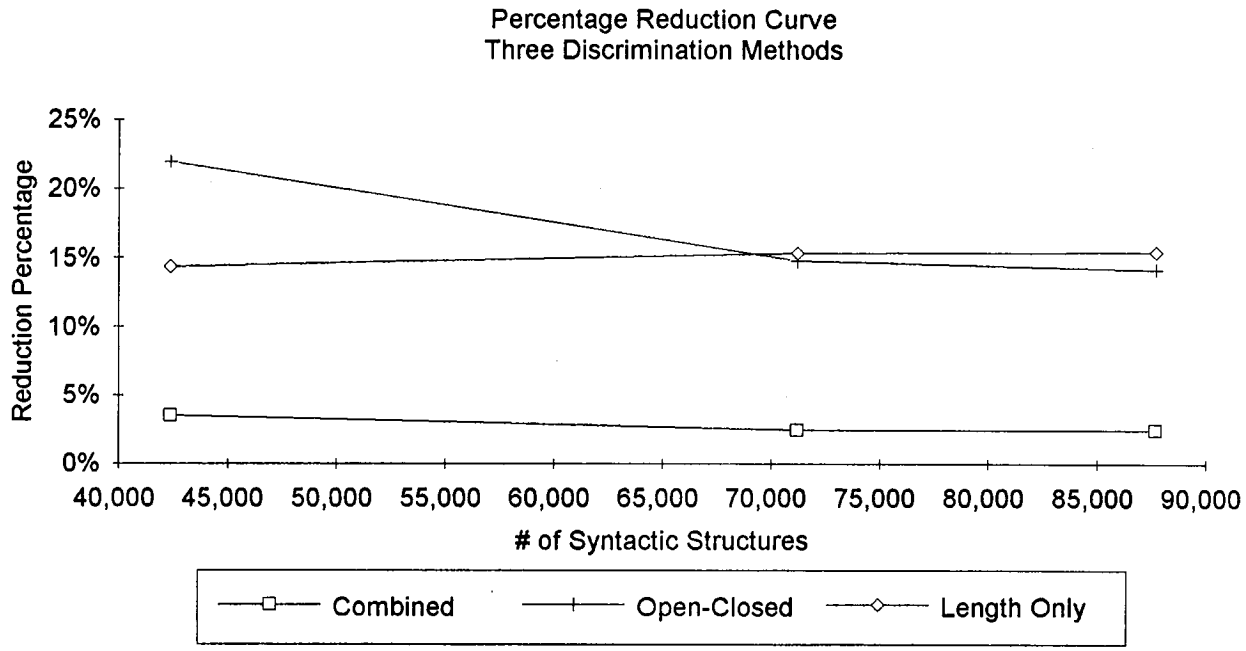

Figure 3

References

[1] Gaudiot, J.-L., Montgomery, C.A. and Stumberger, R.E., "Data-Driven Execution of Natural Language Processing," *Parallel Processing for AI, Workshop W.1, IJCAI-1991*, Sidney, Australia, 1991, pp. 57-62.

[2] Higuchi, T., Kitano, H., Faruya, T., Handa, K., Takahushi, N. and Kokubu, A. "IXM2: A Parallel Associative Processor for Knowledge Processing," *Proceedings Ninth National Conference on Artificial Intelligence*, 1991, pp. 296-303.

[3] Hillis, W.D., *The Connection Machine*, The M.I.T. Press, Cambridge, MA, 1985.

[4] Kitano, H. and Higuchi, T., "Massively Parallel Memory-Based Parsing," *12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, 1991, pp. 918-924.

[5] Kitano, H. and Higuchi, T., "High Performance Memory-Based Translation on IXM2 Massively Parallel Associative Memory Processor," *Proceedings Ninth National Conference on Artificial Intelligence*, 1991, pp. 149-154.

[6] Kitano, H., Moldovan, D. and Cha, S., "High Performance Natural Language Processing on Semantic Network Array Processor," *12th International Joint Conference on Artificial Intelligence*, Sydney, Australia, 1991, pp. 911-917.

[7] Kitano, H., "Massively Parallel Natural Language Processing," *Parallel Processing for AI, Workshop W.1, IJCAI-1991*, Sidney, Australia, 1991, pp. 99-105.

[8] Kitano, H., personal communication.

[9] Nicholl, S. and Wilkins, D.C., "Efficient Learning of Language Categories: The Closed-Category Relevance Property and Auxiliary Verbs," *Proceedings of the Cognitive Science Society*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1990, pp. 455-462.

[10] Riesbeck, C. and Martin, C., "Direct Memory Access Parsing," *Experience, Memory, and Reasoning*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[11] Santos, E. Jr., "A Massively Parallel Self-Timing Context-Free Parser," *Advances in Neural Information Processing Systems I*, D.S. Touretzky (ed.), Morgan Kaufmann Pub., San Mateo, CA, 1989.

[12] Stanfill, C. and Waltz, D., "The Memory-Based Reasoning Paradigm," *Proceedings of the Case Based Reasoning Workshop*, DARPA, 1988.

[13] Stanfill, C. and Waltz, D., "Toward Memory-Based Reasoning," *Communications of the ACM*, 1986.

[14] Stolfo, S.J., Galil, Z., McKeown, K. and Mills, R., "Speech Recognition in Parallel," *Proceedings of the DARPA Speech and Natural Language Workshop*, Morgan Kaufmann Pub., San Mateo, CA, 1989, pp. 353-373.

[15] Tomita, M., *Efficient Parsing for Natural Language*, Kluwer Academic Publishers, 1986.

[16] Winz, S. and Geller, J., "Approaches to Handling Large Grammars in Massively Parallel Parsing Systems," 1993, submitted for publication.

[17] Winz, S., "Massively Parallel Natural Language Parsing", MS Project, CIS Department, New Jersey Institute of Technology, 1992.