A Parallel Parsing Algorithm for Natural Language using Tree Adjoining Grammar*

Tom Nurkkala
nurkkala@cs.umn.edu

Vipin Kumar
kumar@cs.umn.edu

Department of Computer Science, University of Minnesota, Minneapolis, MN 55455

Abstract

Tree Adjoining Grammar (TAG) is a powerful grammatical formalism for large-scale natural language processing. However, the computational complexity of parsing algorithms for TAG is high. We introduce a new parallel TAG parsing algorithm for MIMD hypercube multicomputers, using large-granularity grammar partitioning, asynchronous communication, and distributed termination detection.

We describe our implementation on the nCUBE/2 parallel computer, and provide experimental results on both random and English grammars. Our algorithm delivers the best performance of any TAG parsing algorithm to date, yielding an almost two order-of-magnitude speedup and good efficiency on up to 256 processors.

TAG parsing is a highly unstructured problem. Based on our experience developing a parallel TAG parser, we draw some general conclusions for solving other unstructured problems.

1 Introduction

Automated Natural Language Processing (NLP) will revolutionize our modes of communication with each other and with computers, providing multilingual environments that will fundamentally transform the work of scientists, engineers, scholars, teachers, and students. Hence, NLP has been considered a Grand Challenge application by the Presidential High Performance Computing and Communications Initiative. Some of the key technologies that must be advanced to achieve this revolution include speaker-independent spoken language interfaces, automated machine translation for using documents in different languages, and full-text retrieval systems to provide access to the vast emerging databases of text and multimedia information.

Critical to the success of such systems is a robust and wide-coverage model for natural language. One such language model is Tree Adjoining Grammar (TAG). TAGs can express complex linguistic phenomena more naturally than can context-free grammars, the underlying language model employed in most existing natural language systems. Tree Adjoining Grammar was introduced by Joshi, Levy, and Takahashi [7], who show that the languages generated by TAGs are in the class of mildly context-sensitive languages, strictly between context-free and context-sensitive languages. It is this property of TAGs that give them their expressive power. Several articles provide examples of the use of TAGs in natural language processing [8, 6, 3, 1].

The expressive power of TAG comes at increased parsing cost. Although context-free grammars can be parsed in $O(n^3)$ time (for input strings of length n), general TAGs require time $O(n^6)$. This complexity is prohibitive for even moderate length input strings. Thus, TAG parsing algorithms are good candidates for parallelization.

Vijay-Shankar and Joshi [17] presented the first feasible sequential parsing algorithm for TAGs. Their parser is a bottom-up, CYK-style parser [2] with a time complexity of $\Theta(n^6)$. A drawback of this parser is that it requires TAGs in normalized form, similar to Chomsky normal form for context-free grammar. A parallel parsing algorithm for TAGs was introduced by Palis, Shende, and Wei [12]. Their algorithm runs in linear time on the length of the input string, but requires $O(n^5)$ processors in a systolic array, and only works with normalized TAGs.

More recently, Palis and Wei [14] have designed a new parsing algorithm for general TAGs. In previous work, grammar size (G) was considered a constant, and parsing complexity was expressed in terms of input string length (n). This makes sense for program compilation, where typically $n \gg G$. However, in nat-

*This work was partially supported by IST/SDIO through the Army Research Office grant #28408-MA-SDI and by the Army High Performance Computing Research Center at the University of Minnesota.

ural language processing, grammars are large (thousands of rules) and sentences short (typically 30 words or fewer); that is, $n \ll G$. Palis and Wei's algorithm is designed to work well under the latter condition. It runs in time $O(nL^2)$, where L is a parameter capturing the interdependence between grammar and input string. L can vary from a constant to $O(n^4)$, yielding a run time between $O(n)$ and $O(n^9)$. Typically, the algorithm runs in $O(n^5)$ time, and thus is superior on average to Vijay-Shankar and Joshi's algorithm.

Palis and Wei have introduced a SIMD parallel implementation of their parser on the Connection Machine CM-2 [13, 14]. Giving speedup results for various grammar sizes, they show that their algorithm has the expected asymptotic behavior relative to G. However, their data also show a speedup saturation, and hence an upper bound on efficiency. This bound is due to a fine-granularity mapping scheme that leaves a significant fraction of the CM-2 processors idle. Much of the execution time of the algorithm is spent performing set operations (on ***index sets***—see Section 3). Palis and Wei use the CM-2 as a two-dimensional mesh, mapping each set to a row of the mesh, with each set element allocated to a processor in the row. However, the cardinality of a set is often smaller than the number of processors in a row of the mesh, leaving many of the processors idle. This results in a bound on efficiency—on a 16,384-processor CM-2, they report a speedup of only 40 with respect to a Sun 4 workstation.

This paper describes a new parallel formulation of Palis and Wei's TAG parser. Our algorithm for a hypercube-connected MIMD multicomputer differs from theirs in three ways. First, it partitions the TAG at a higher level of granularity, mapping a portion of the grammar to every processor. Thus, our algorithm does not have an *a priori* limit on efficiency due to the mapping. Second, our algorithm avoids much of the idling overhead due to synchronous operation by asynchronously distributing intermediate parsing information throughout the multicomputer as it is generated by each processor. Third, our algorithm uses an asynchronous distributed termination detection algorithm, which runs concurrently with the parse.

An expanded version of this paper is available from the University of Minnesota Department of Computer Science as Technical Report 93-31.

2 Tree Adjoining Grammar

A Tree Adjoining Grammar consists of two finite, disjoint sets of terminal and non-terminal symbols, and two sets of trees, ***initial trees*** and ***auxiliary trees***, collectively referred to as ***elementary trees***. The nodes of elementary trees are generically referred to as ***grammar nodes***. One non-terminal symbol is distinguished as the ***start symbol*** of the grammar.

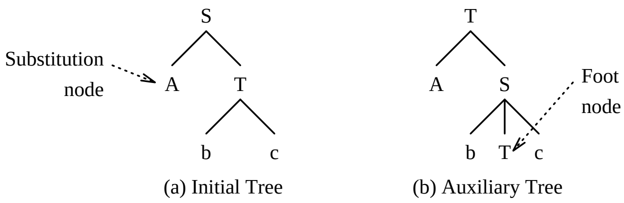

Figure 1: Types of TAG trees

In each elementary tree (Figure 1), interior nodes are labeled by non-terminals, and leaf nodes are labeled by either terminals or non-terminals. We show non-terminals as upper-case letters or strings, and terminals as lower-case letters or strings. Leaf nodes labeled by non-terminals are called ***substitution nodes***. In an auxiliary tree, exactly one of its leaf nodes (the ***foot node***) must be labeled by the same non-terminal that labels its root. In natural language processing applications, initial trees represent basic categories such as noun phrases, verb phrases, and sentences. Auxiliary trees encode modifiers, such as adjectives, adverbs, and relative clauses.

Elementary trees are composed by two operations: ***substitution***, which uses initial trees and ***adjunction***, which uses auxiliary trees. The resulting tree is a ***derived tree***, which can likewise be composed with other trees. Substitution (Figure 2) replaces a substitution node with an initial tree whose root has the same label as the substitution node.

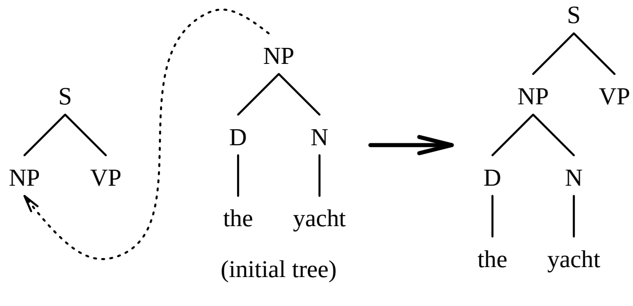

Figure 2: Substitution

Adjunction (Figure 3) inserts an auxiliary tree inside another tree. Adjunction takes place at an internal node (the ***adjunction site***) whose label is identical to that of the auxiliary tree's root node. Its effect is to create a new tree in which the sub-tree rooted at the adjunction site is replaced by the auxiliary tree, and the excised sub-tree is reattached at the foot node of the auxiliary tree.

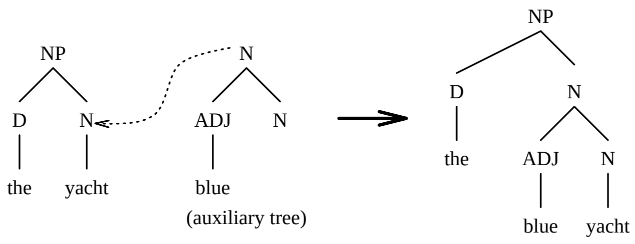

Figure 3: Adjunction

Given a TAG and an input string, a TAG parsing algorithm applies a series of substitution and adjunction operations in an attempt to compose a derived tree whose root is labeled by the start symbol, and whose leaves consist of exactly the input string. If such a tree can be composed, the TAG ***accepts*** the input string; otherwise, it ***rejects*** it.

3 Sequential Algorithm

Our parsing algorithm is based on the the sequential algorithm presented by Palis and Wei [13, 14]. It takes as input a TAG G and an input string w, where w is an n-word string of terminals $a_1 \cdots a_n$, and returns a boolean value indicating whether G accepts w.

The principle activity of the algorithm is maintaining ***index tuples*** (or simply ***tuples***), which are 4-tuples of the form $(i, j, k, l), 0 \leq i \leq j \leq k \leq l \leq n$. An ***index set*** is a set of index tuples; one such set is associated with each grammar node. For a grammar node α, its index set is denoted $IS(\alpha)$. The index set of the root node of an elementary tree is its ***root index set***.

The components of an index tuple correspond to the portion of the input string spanned by the tuple. That is, for a tuple t and grammar node α, $t \in IS(\alpha)$ if and only if there is a derivation from α of the portion of the input string corresponding to the components of t. Consider the input string, "the boy sailed the blue yacht." Each position in the string can be numbered as follows:

$$^{0}\text{the}^{1}\text{boy}^{2}\text{sailed}^{3}\text{the}^{4}\text{blue}^{5}\text{yacht}^{6} \qquad (1)$$

A tuple indicating that "the blue yacht" is spanned is $(3, 4, 4, 6)$.[1] Because the foot nodes of auxiliary trees are labeled by non-terminals, tuples must capture cases where two portions of the input string, separated by the foot node, are spanned. For example, "the boy ... the blue yacht" would be represented as $(0, 2, 3, 6)$. A tuple in the index set of a node α indicates that α is known to span at least the corresponding portion of the input. Refer to Figure 4, where "Y" represents the foot node spanned by α. Note that in any tuple where $j = k$, a foot node is not spanned.

[1] Actually, any tuple $(3, k, k, 6)$, $3 \leq k \leq 6$ would indicate the same sequence of input symbols.

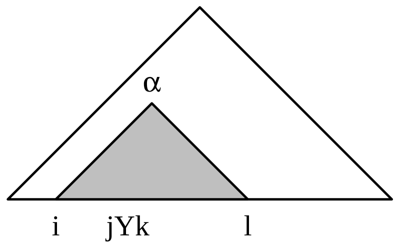

Figure 4: Index Tuple $(i, j, k, l) \in IS(\alpha)$.

The basic sequential parser is shown as Algorithm 1. Initially, all index sets are empty. In step 1, tuples are added to the index sets of leaf nodes of the grammar whose labels match terminals in the input string. For example, given input string 1, above, all such nodes labeled "the" would have $(0, 0, 0, 1)$, and $(3, 3, 3, 4)$ added to their index sets.

1. initialize
2. **repeat** for all grammar nodes
   until no index set changes
   substitution
   adjunction
   concatenation
   completion
3. recognize

Algorithm 1: Sequential TAG parsing algorithm

In step 2, the algorithm percolates index tuples bottom-up (see below), until all index sets stop changing. Finally, in step 3, the parser checks all root index sets labeled with the start symbol of the grammar. If any such set contains a tuple of the form $(0, k, k, n)$, $0 \leq k \leq n$, the start symbol spans the entire input string, and the input is accepted. If no such tuple is found, the input is rejected.

On each iteration of step 2 (the ***parsing loop***), the algorithm considers all grammar nodes α, and performs one of four basic operations. Each of these operations involves a set union into a node's index set. We refer to this union as ***augmenting*** a set. Note that these are simplified descriptions of the operations; they are:

1. ***Substitution***. For a substitution node α, aug-

ment $IS(\alpha)$ with the root index sets of all initial trees with the same label as α.

2. ***Adjunction***. For an internal node α, augment $IS(\alpha)$ with the a new set of tuples formed from the root index set of each auxiliary tree, according to appropriate restrictions for adjunction.

3. ***Concatenation***. Augment $IS(\alpha)$ with a new set of tuples formed from the index set of the left-most sibling of α, according to appropriate restrictions for concatenation.

4. ***Completion***. Augment $IS(\alpha)$ with the index set of its right-most child.

In summary, the parsing algorithm begins by assigning index tuples to those leaf nodes whose labels correspond to symbols in the input string. Tuples percolate through elementary trees left-to-right at the same depth by concatenation, and bottom-to-top one level at a time by completion. As tuples reach the roots of initial trees, they are added to the tuple sets of identically labeled substitution nodes during substitution. Similarly, as tuples reach the roots of auxiliary trees, they are added to the tuples sets of appropriate internal nodes during adjunction. Tuples added by either substitution or adjunction move through elementary trees left-to-right and bottom-to-top by concatenation and completion, as above. Finally, when no index set changes in an iteration of the parsing loop, the recognition test checks the index sets of all root nodes labeled with the start symbol, searching for index tuples spanning the entire input string.

4 Parallel Algorithm

As outlined in Section 3, the principle activity of the TAG parser is manipulating index sets within each elementary tree. The parser achieves parallelism by partitioning elementary trees among the processors. Each processor is responsible only for the computations associated with the elementary trees it stores locally (its ***local trees***). Let p be the number of processors and G be the number of elementary trees in a TAG. Then each processor is assigned G/p elementary trees in round-robin fashion.

A parsing loop similar to that in Algorithm 1 performs the parse by updating index sets. However, in the parallel formulation, the loop operates only on local trees. Since an entire tree is stored locally, this is sufficient for the concatenation and completion operations, which require only index tuples from other nodes in the same tree.

However, the substitution and adjunction operations may require the root index sets of elementary trees maintained by other processors. Thus, each processor stores a copy of the root index set of all elementary trees. Whenever a tuple is added to a local root index set, it is placed on the local ***tuple queue***, a list of tuples waiting to be broadcast to other processors. Each processor periodically broadcasts its tuple queue to all others, allowing them to update their copies of the local processor's root index sets.

A finer-grained partitioning could be implemented by splitting elementary trees among different processors. Such a partitioning may improve work load balance by distributing an active tree across several processors. However, it will considerably increase communication overhead, since some concatenation and completion operations will involve non-local grammar nodes. Thus, we have not pursued a tree-splitting approach.

In the sequential algorithm, the parsing loop terminates when all index sets stop changing. However, in the parallel algorithm, index sets are distributed among the processors, so local inactivity does not imply global termination. For example, when processor P_i has detected local termination, P_j may have just assigned a new tuple to the root index set of an initial tree having the same label as a substitution node on P_i. This new tuple must be broadcast to P_i to allow substitution, reactivating P_i. Global termination is achieved only when *all* processors have become quiescent. Since new tuples are broadcast asynchronously by each processor, the algorithm uses a distributed termination detection algorithm.

4.1 Hypercube Implementation

Algorithm 2 is our parallel TAG parsing algorithm as implemented on a hypercube-connected MIMD multicomputer. The initial static allocation of elementary trees takes time $O(G)$, for a TAG with G elementary trees. We assume the grammar remains fixed for the parse of a number of input strings, each of which can be broadcast to all processors in time $O(n \lg p)$, for input strings of length n.

The one-to-all broadcast of root index tuples in step 5 is performed in time $O(m \lg p)$, where m is the size of the tuple queue, using a tree whose root is mapped to the local processor. This tree-based propagation relies on step 4, which forwards tuple queues to their proper destinations based on their processor of origin. Note that there are p one-to-all broadcast trees operating simultaneously, one rooted at each of the p processors of the multicomputer. The performance of the algorithm is relatively insensitive to the

1. initialize
2. **while** global termination not detected
3. **for** each local grammar node perform
   substitution
   adjunction
   concatenation
   completion
4. **if** tuple alarm
   propagate tuples
5. broadcast tuple queue
6. **while** no local index sets change
   and global terminate not detected
7. wait for incoming tuples
   or global termination
8. recognize

Algorithm 2: Parallel TAG parsing algorithm

delay used to signal tuple propagation, because tuple communication is relatively infrequent. Values from 100 μs to 100 ms yield similar performance.

Global termination is implemented using Dijkstra's distributed termination detection algorithm [4]. This algorithm circulates a termination token around a ring mapped onto the hypercube. The token circulates concurrently with parsing and one-to-all broadcast activity.

Our algorithm improves performance by using a dirty bit on each index set, indicating whether it has been modified since the last time through the parsing loop. When an index set operation is about to be performed, if both sets are clean, the operation can be skipped because it will yield the same result as its previous execution.

5 Experimental Results

We have implemented Algorithm 2 on the nCUBE/2, a hypercube-connected MIMD parallel computer. Our experiments were conducted with both randomly generated grammars and an English grammar provided by the University of Pennsylvania. Random grammars allowed us to control the number and size of elementary trees, and the size of the input string. Use of an English grammar demonstrated the applicability of our parser to natural languages. In the performance data that follow, p is the number of processors, T_p is the parallel run-time in seconds on p processors, $S = T_1/T_p$ is the speedup, and $E = S/p$ is the efficiency.

5.1 Performance

5.1.1 Random Grammars

Table 1 shows the performance of the algorithm with a grammar of 1,024 random elementary trees (eight nodes per tree average), parsing a 20-word input string. (These data are comparable to those of Palis and Wei [13, 14].) Note that since the memory size of a single node of the nCUBE/2 is insufficient to store the number of index tuples generated by this large grammar, the value for $p = 1$ is estimated, based on the time spent performing useful parsing activity for $p = 4$.

p	T_p	S	E
1	2520.910	1.000	1.000
4	647.099	3.896	0.974
8	324.024	7.780	0.973
16	173.054	14.567	0.910
32	100.767	25.017	0.782
64	63.592	39.642	0.619
128	46.307	54.439	0.425
256	33.060	76.253	0.298

Table 1: Performance on a 1,024-tree random grammar

These data show fairly high efficiencies across a range of processor sizes, giving roughly a two-order-of-magnitude speedup on 256 processors. Note that for 256 processors, T_p is approximately 17 times faster than the best previous performance on this size grammar and input string. (Palis and Wei achieved $T_p = 568$ on a 16,384-processor CM-2).

5.1.2 English Grammar

Table 2 shows the performance of the parser on a small English grammar. This grammar has 225 elementary trees, and the results shown are the total time to parse 22 input sentences (average length 7.5 words, 16 words in longest).

Figure 5 plots the efficiency values from Table 2 against similar values for a 256-tree random grammar. These curves suggest that the algorithm performs similarly both on random and on English grammars. Thus, our other results for large random grammars should hold for large English grammars as well.

5.2 Analysis

5.2.1 Computation

Tuple index sets are unordered, so the number of comparisons required to form the union of index sets S_1

p	T_p	S	E
1	4.538	1.000	1.000
2	2.613	1.736	0.868
4	1.373	3.305	0.826
8	0.735	6.170	0.771
16	0.423	10.715	0.670
32	0.280	16.222	0.507
64	0.238	19.055	0.298
128	0.270	16.818	0.131

Table 2: Performance on a 225-tree English Grammar

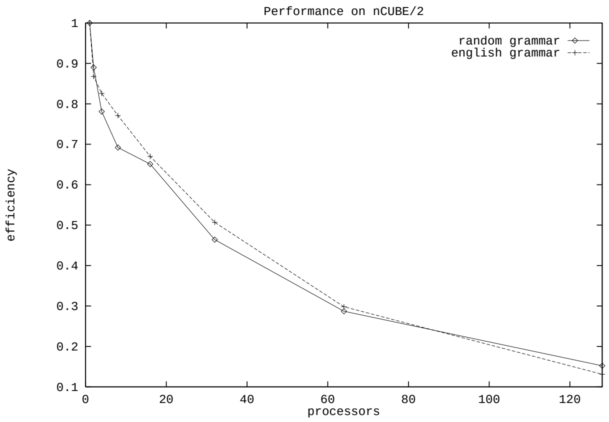

Figure 5: Performance comparison between English and random grammars of similar size

and S_2 is given by $|S_1| \times |S_2|$. Since each operation in the algorithm's parsing loop involves a union of index sets (Section 3), the total number of comparisons is a good measure of the amount of useful work done by the algorithm.

Figure 6 shows a plot of total work versus sentence length for various grammar sizes. Note that sentence length has a much greater impact on work load than grammar size. This is to be expected, since the algorithm's complexity is much more sensitive to sentence length (Section 1).

5.2.2 Overhead

The principle sources of overhead in the algorithm are tuple set communication and processor idling.

Overhead Due to Communication. Figure 7 is a 3D plot of communication activity during a parse. Early activity (first five seconds) is due to grammar and input string distribution. Except for the final spike of activity (due to collection of results), all remaining communication results from root index tuple

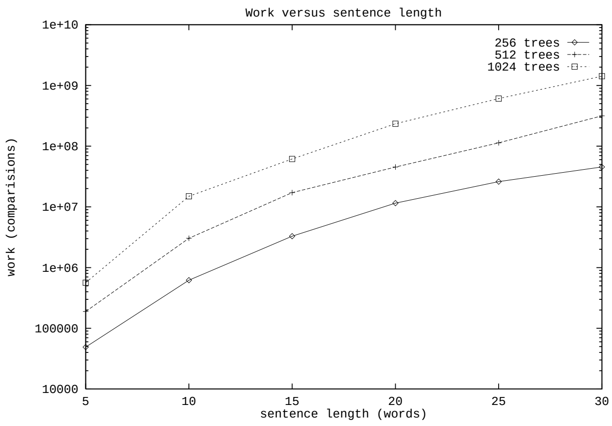

Figure 6: Work versus sentence length

propagation. Recall that root index tuples are broadcast to all processors, giving rise to activity across the multicomputer.

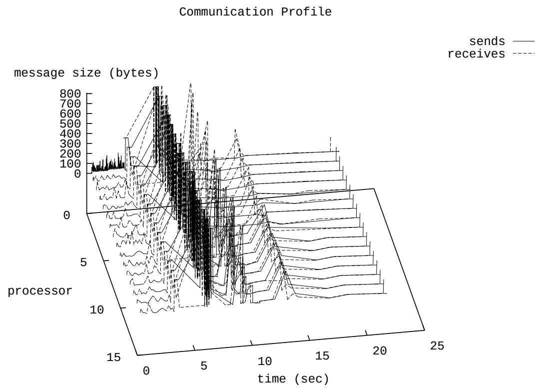

Figure 7: Communication profile for parsing a 20-word input string using a 256-tree grammar.

Since only new root index tuples are broadcast, communication drops off as tuple sets begin to stabilize. The flurry of early activity (at about nine seconds) occurs when tuples assigned during parse initialization reach the root nodes of their trees (likely through the completion operation). The remaining spikes occur when tuples generated during parsing reach the root nodes of their new trees.

Figure 8 shows a plot of the percentage overhead due to communication for various sentence lengths. Since execution time varies directly with sentence length (Figure 6), and most communication is done early in the parse (Figure 7), communication overhead is more significant for shorter sentences, whose parses have shorter duration. As sentence length increases, more work is performed within each tree (and hence, within each processor), resulting in less communication overhead.

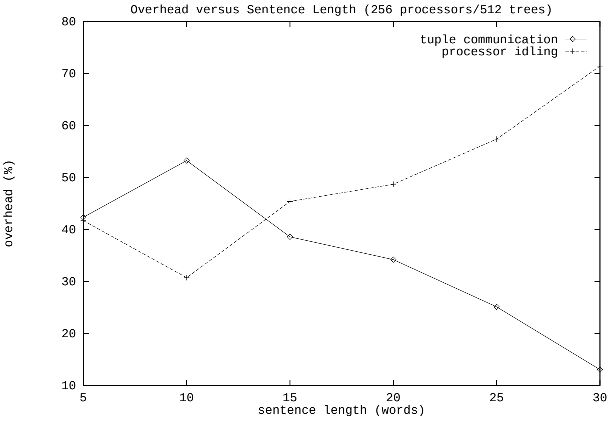

Figure 8: Percentage of overhead due to communication and idling for various length input strings.

Overhead Due to Processor Idling. During a parse, the work in a given elementary tree varies significantly and can move to another tree in unpredictable ways as substitution and adjunction occur. Since elementary trees are distributed across the multicomputer, processor load can vary in the same unpredictable fashion. This variation yields work load imbalances among processors. While local trees at processor P_i may have a large number of tuples being processed in the parsing loop, processor P_j may have very few. Thus, P_j may be idle for a significant fraction of the time that P_i is performing useful work.

Figure 8 also illustrates the percentage overhead due to idling. Most parsing activity takes place early in the parse, when new tuples are being broadcast throughout the multicomputer. After index sets begin to settle, a significant fraction of the processors become idle. Thus, for parses of longer duration (i.e., those of longer sentences), load imbalance tends to become worse and idling becomes the most significant source of overhead.

The figure also shows a crossover point where idling begins to dominate the algorithm's overhead. Similar data show that the crossover point shifts toward smaller inputs as the grammar size decreases. This shift reflects the decreased communication required for smaller grammars, which have fewer root index tuples to broadcast.

5.2.3 Scalability

In many parallel algorithms, more work implies greater concurrency and higher performance [9, 5]. Such algorithms are *scalable* in that they deliver good performance on larger numbers of processors provided the work load is increased sufficiently. In this algorithm, increased work is due to two factors: grammar size and sentence length.

Effect of Grammar Size. Figures 9 and 10 are plots of speedup (S) and efficiency (E) versus number of processors for parsing a 30-word input string with grammars of various sizes. These plots show that the algorithm is scalable with respect to grammar size, since performance improves as grammar size increases.

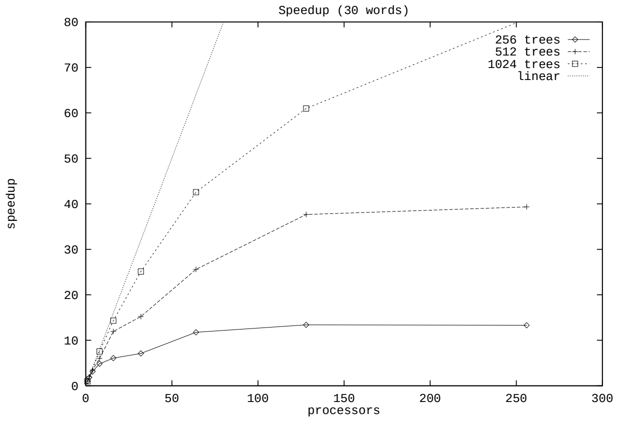

Figure 9: Speedup for a 30-word input sentence

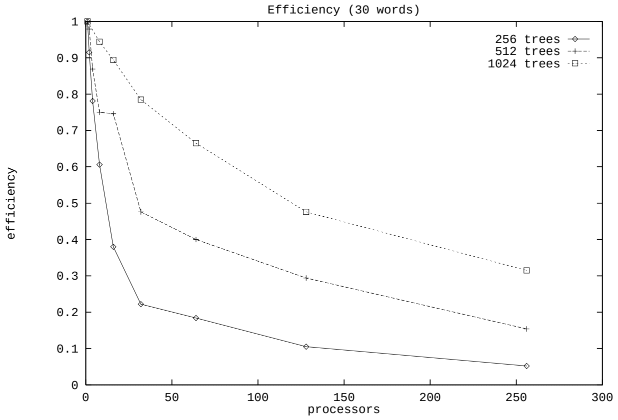

Figure 10: Efficiency for a 30-word input sentence

Effect of Sentence Length. Figure 11 shows a plot of efficiency versus number of processors for various sentence lengths. The algorithm is not scalable with respect to sentence length, since it achieves no gain in efficiency for longer sentences, even though such sentences entail considerably more work (Figure 6).

The haphazard variation in efficiency for various length sentences results from the unpredictable overhead arising from the combination of communication and idling (Section 5.2.2). For long sentences, poor efficiency results from significant idling in the later stages of parsing. Adopting a dynamic load balancing scheme (Section 6.1) may improve scalability relative to sentence length by reducing idling late in the parse. However, load balancing may itself become a significant source of overhead, because it requires sending elementary trees, together with their index sets. Late in a parse, these sets can grow quite large.

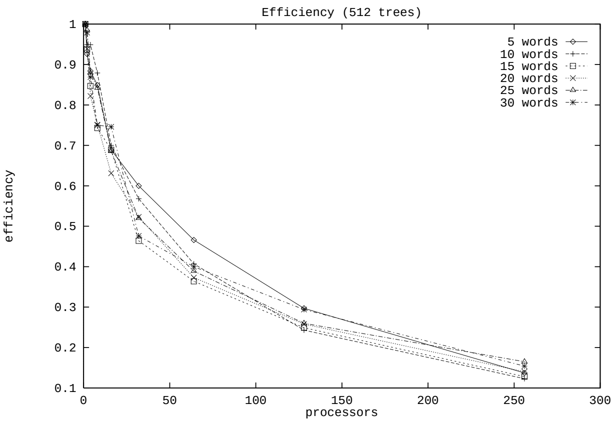

Figure 11: Efficiency variation for various sentence lengths

6 Future Work

Future work on parallel TAG parsing should follow two lines. First, using dynamic load balancing to reduce idling, the major cause of reduced efficiency. Second, exploring an Earley-style parsing algorithm with better average-case complexity than the current parser.

6.1 Dynamic Load Balancing

Using ***dynamic load balancing***, when a processor becomes idle, it requests work from other processors in the ensemble. The processor that receives the request determines whether it has sufficient work (i.e., pending tuple operations). If so, it sends work to the requesting processor in the form of elementary tree(s) and index sets.

The problem of determining when a processor has sufficient work to send is a thorny one. Sending a tree and its index sets (which may be large, especially late in the parse) is itself a source of overhead. Thus, the tree(s) transmitted must result in sufficient *future* work to reduce the overhead due to idling by more than the overhead of transmitting the tree(s).

One way to estimate future work is to assume that if a tree has been busy in the past, it will likely be busy in the future. This is a poor metric—if all tuple propagation has completed in a tree, it will have large past work, but no further activity will take place.

A better predictive metric is a *structural* one. New work is generated in a tree when substitution or adjunction takes place at one of its nodes. At a node that is low and to the left in a tree, this will result in more work than at one high and to the right, since concatenation and completion propagate tuples left-to-right and bottom-to-top. Thus, future work can be predicted by noting *when* adjunction or substitution is performed in a tree, and *at what node*.

6.2 Earley-Style Algorithm

Schabes [15] introduces an Earley-style parsing algorithm [2] for general TAGs. The algorithm exhibits an asymptotic behavior of $O(n^6)$, but has better average-case complexity because it uses top-down prediction as well as bottom-up parsing. Thus, a parallel version of this algorithm should yield improved performance relative to the parser described here.

Furthermore, Schabes and Joshi [16, 15] detail a variant of TAG called Lexicalized Tree Adjoining Grammar (LTAG). In LTAG, each elementary tree contains at least one leaf node labeled with a terminal symbol called its ***anchor***. For a given input string, only those trees whose anchor appears somewhere in the input can possibly participate in the parse. Other trees will never generate tuples that span their entire yield because at least their anchor node's index set will remain empty. This lexical information permits a (potentially large) subset of the elementary trees to be deactivated prior to the parse. This should result in better raw performance (i.e., smaller T_p), since fewer index sets need to be processed in each iteration of the parsing loop.

Our current work focuses on parallel formulations of LTAG [10]. Early results on the KSR1 parallel computer at the University of Florida show promising results—80% efficiency on 32 processors for a moderate size grammar and input string [11].

7 Conclusions

We have introduced a new parallel algorithm for natural language parsing using Tree Adjoining Grammar. Our MIMD implementation delivers good speedup and efficiency on both random and English grammars. It delivers the best performance of any TAG parser to date.

We have come to view TAG parsing as a prototypical example of a large class of highly unstructured problems yet to be solved using large-scale multicomputers. Generalizing our experience with TAG parsing, parallel algorithms for unstructured problems will exhibit (1) sources of overhead having a non-uniform impact on performance throughout execution, (2) sources of overhead interacting in complex and unexpected ways, and (3) significant overhead due to idling because of processor load imbalances.

Acknowledgments

George Karypis at the University of Minnesota provided invaluable assistance both on the algorithm described here and on this paper. Our sincere thanks to Michael Palis at the New Jersey Institute of Technology, Aravind Joshi and Bangalore Srinivas at the University of Pennsylvania, and Yves Schabes at the Mitsubishi Electric Research Laboratories. We also gratefully acknowledge Robert Benner and Sudip Dosanjh at Sandia National Laboratories for the use of their 1024-processor nCUBE/2.

References

[1] A. Abeillé. A lexicalized tree adjoining grammar for French: The general framework. Technical Report MS-CIS-88-64, Department of Computer and Information Science, University of Pennsylvania, 1988.

[2] A. Aho and J. Ullman. *The Theory of Parsing, Translation, and Compiling, Vol. 1: Parsing.* Prentice-Hall, 1982.

[3] K. Bishop, S. Cote, and A. Abeillé. A lexicalized tree adjoining grammar for English. Technical report, Department of Computer and Information Science, University of Pennsylvania, 1989.

[4] Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16:217–219, June 1983.

[5] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency function: A scalability metric for parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1993.

[6] A. K. Joshi. How much context-sensitifity is necessary for characterizing structural descriptions—tree adjoining grammars. In D. Dowty, L. Karttunen, and A. Zwicky, editors, *Natural Language Proessing—Theoretical, Computational and Psychological Perspectives.* Cambridge University Press, 1985.

[7] A. K. Joshi, L. S. Levy, and M. Takahashi. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10:136–63, 1975.

[8] A. S. Kroch and A. K. Joshi. The linguistic relevance of tree adjoining grammars. Technical Report MS-CIS-85-16, University of Pennsylvania, Department of Computer and Information Science, April 1985.

[9] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin/Cummings, 1994.

[10] Tom Nurkkala. *Parallel Algorithms for a Highly Unstructured Problem: Natural Language Parsing using Tree Adjoining Grammar.* PhD thesis, University of Minnesota, 1994 (to appear).

[11] Tom Nurkkala and Vipin Kumar. The performance of a highly unstructured parallel algorithm on the KSR1. In *Proceedings of the Scalable High Performance Computing Conference.* 1994 (to appear).

[12] Michael A. Palis, Sunil Shende, and David S. L. Wei. An optimal linear-time parallel parser for tree adjoining languages. *SIAM Journal on Computing*, 19(1):1–31, February 1990.

[13] Michael A. Palis and David S. L. Wei. Parallel parsing of tree adjoining grammars on the connection machine. Technical report, University of Pennsylvania, 1991.

[14] Michael A. Palis and David S. L. Wei. Massively parallel parsing algorithms for natural language. In L. Kanal, Vipin Kumar, Christian Suttner, and H. Kitano, editors, *Parallel Processing for Artificial Intelligence.* North-Holland, 1993.

[15] Yves Schabes. *Mathematical and Computational Aspects of Lexicalized Grammars.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, August 1990.

[16] Yves Schabes and A. K. Joshi. An earley-type parsing algorithm for tree adjoining grammars. Technical Report MS-CIS-88-36, University of Pennsylvania, Department of Computer and Information Science, June 1988.

[17] K. Vijay-Shankar and A. K. Joshi. Some computational properties of tree adjoining grammars. In *Proceedings of the 11th Meeting of the Association of Computational Linguistics*, August 1986.