TAL PARSING IN $o(N^6)$ TIME*

SANGUTHEVAR RAJASEKARAN†

Abstract. In this paper we present algorithms for parsing general Tree Adjoining Languages (TALs). Tree Adjoining Grammars (TAGs) have been proposed as an elegant formalism for natural languages. It was an open question for the past ten years as to whether TAL parsing can be done in time $o(n^6)$. We settle this question affirmatively by presenting an $O(n^3\ M(n))$ time algorithm, where $M(k)$ is the time needed for multiplying two Boolean matrices of size $k \times k$ each. Since $O(k^{2.376})$ is the current best known value for $M(k)$, the time bound of our algorithm is $O(n^{5.376})$. On an EREW PRAM our algorithm runs in time $O(n \log n)$ using $\frac{n^2\ M(n)}{\log n}$ processors. In comparison, the best known previous parallel algorithm had a run time of $O(n)$ using n^5 processors (on a systolic array machine).

We also present algorithms for parsing Context Free Languages (CFLs) and TALs whose worst case run times are $O(n^3)$ and $O(n^6)$, respectively, but whose average run times are better. Therefore, these algorithms may be of practical interest.

Key words. Tree Adjoining Languages, Parsing, Natural Language Processing, Parallel Algorithms, Context Free Grammars.

AMS subject classifications. 11Y16, 68N20, 68Q20, 68Q22, 68Q25, 68S05, 68U30.

1. Introduction.

In [9], Joshi, Levy, and Takahashi introduced a grammatical formalism called Tree Adjoining Grammars (TAGs). TAGs are strictly more expressive than context free grammars (CFGs). For instance, $\{a^n b^n c^n | n \geq 0\}$ can be generated with a TAG but this language is not context free. TAGs have been shown to be good grammatical systems for natural language [10]. In this paper we will restrict our discussion only to the problem of language recognition, since such an algorithm can also be used for retrieving a parse. We use the terms 'parsing' and 'recognition' interchangeably.

Many papers have been written on the parsing problem for TAGs. Vijayashanker and Joshi [19] gave the first polynomial time algorithm for TAL parsing. Their algorithm had a run time of $O(n^6)$, assuming that the size of the grammar is constant. This assumption has been made in most of the literature on parsing. This algorithm had a flavor similar to that of the Cocke-Kasami-Younger (CKY) algorithm for CFL parsing.

An Earley-type parsing algorithm has been given by Schabes and Joshi [16]. This algorithm also takes $O(n^6)$ time. For some special cases of TALs, better algorithms have been discovered [15]. Several attempts have been made in the past to obtain $o(n^6)$ time algorithms (see e.g., [5, 6]). The parallel complexity of TAL parsing has been studied as well. As an example, Palis, Shende, and Wei present an optimal linear time algorithm on a systolic array machine with n^5 processing elements [12]. An excellent treatise on TAGs can be found in [13]. Recently Nurkkala and Kumar [11] presented an efficient $O(n^6)$-work parallel algorithm for TAL parsing.

It was an open question since 1986 [19] as to whether general TAL parsing can be done in $o(n^6)$ time. In this paper we present an algorithm that runs in time $O(n^3\ M(n))$, where $M(k)$ is the time needed for multiplying two Boolean matrices of size $k \times k$ each. The best known value for $M(k)$ is $O(k^{2.376})$ as has been shown

* This research was supported in part by an NSF Research Initiation Award CCR-92-09260, an NSF Award CCR-95-03007, and an ARO grant DAAL03-89-C-0031.

† Department of Computer and Information Science, University of Florida, Gainesville, FL 32611 (raj@cis.ufl.edu).

by Coppersmith and Winograd [2]. In a related work, Valiant [18] has shown that CFL parsing can be reduced to Boolean matrix multiplication. Similar work for CFL parsing has also been done by Graham, Harrison, and Ruzzo [4]. Though we make use of algorithms for Boolean matrix multiplication, our approach is different from Valiant's. Direct application of Valiant's technique to TAL parsing seems to fail [14, 15].

Our TAL parsing algorithm also runs in $O(n \log n)$ time using $\frac{n^2\ M(n)}{\log n}$ EREW PRAM processors. This run time is nearly the same as that of the previously best known algorithm for TAL parsing [12]. The processor bound of our algorithm is significantly better than that of [12]. Though we make use of a different model than that of [12], for example one could invoke Ranade's PRAM simulation algorithm to infer that it is possible to obtain nearly the same run time on the systolic array machine, nearly preserving the work done.

2. Definition of TAGs. A TAG is a 5-tuple $G = (N,\ \Sigma \cup \{\epsilon\},\ I,\ A,\ S)$, where

N is a finite set of nonterminal symbols,
Σ is a finite set of terminal symbols disjoint from N,
ϵ is the empty terminal string not in Σ,
I is a finite set of labelled *initial trees*,
A is a finite set of labelled *auxiliary trees*, and
$S \in N$ is the distinguished start symbol.

Initial and auxiliary trees of a TAG are *elementary trees*. All internal nodes of elementary trees are labelled by nonterminal symbols. Every initial tree is labelled at the root by the start symbol S and has leaf nodes labelled by symbols from $\Sigma \cup \{\epsilon\}$. An auxiliary tree has both its root and exactly one leaf node (called the *foot node*) labelled by the same nonterminal symbol. All other leaf nodes are labelled by symbols from $\Sigma \cup \{\epsilon\}$. An example of a TAG is given in Figure 1.

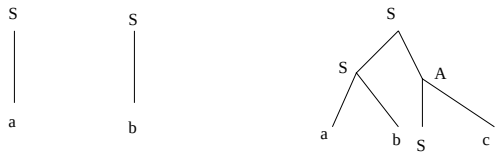

FIG. 1. *TAGs: An Example*

An operation called *adjunction* composes trees of the grammar as follows: Let γ be a tree containing some internal node labelled X, and let β be an auxiliary tree whose root is also labelled by X. Adjoining β into γ results in the tree α (see Figure 2). The formalism also supports *constrained adjunction, selective adjunction, and obligatory adjunction*. See e.g., [19].

Figure 4 displays a TAG that generates the language $L = \{a^n b^n c^n | n \geq 1\}$. Using the pumping lemma for CFGs, we could readily verify that L is not context free. Thus TAGs are strictly more powerful than CFGs. Also, TAGs possess numerous interesting linguistic properties. The linguistic significance of TAGs is described in [8] and [10].

Joshi [8] showed how TAGs factor recursion and the domain of dependencies in an elegant way. Since the introduction of TAGs, several other formalism have been proposed for natural languages. Examples include the Linear Indexed Grammars [3] and Combinatorial Categorial Grammars [17]. These formalisms are different in

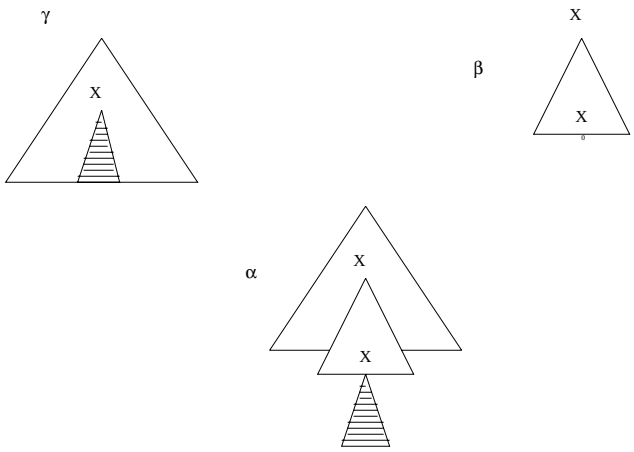

FIG. 2. *The Operation of Adjunction*

terms of the formal objects and operations defined and were motivated by different aspects of language structure. Surprisingly, all these formalisms have been shown to be equivalent thereby increasing the possibility that these formalisms capture some of the important and fundamental aspects of languages.

2.1. Organization of this paper. The rest of this paper is organized as follows: In Section 3 we provide an overview of Vijayashanker-Joshi's algorithm for TAL parsing. In Section 4 we present our algorithm for CFL parsing and show how this algorithm can be extended to TALs to obtain a run time of $O(n^6)$. We apply matrix multiplication algorithms in Section 5 to show that TAL parsing can be done in time $O(nM(n^2))$. We reduce this run time further to $O(n^3M(n))$ in Section 6. Finally, Section 7 concludes the paper.

3. Vijayashanker-Joshi's Algorithm. In this section we provide a brief summary of Vijayashanker-Joshi's algorithm [19]. Their algorithm is similar to the CKY algorithm for CFL parsing. If $a_1a_2 \ldots a_n$ is any given input, CKY algorithm for CFL parsing constructs a two dimensional array A such that $A[i,j]$ is the set of all nonterminals that can derive the substring $a_{i+1}a_{i+2} \ldots a_j$. Array A can be constructed in $O(n^3)$ time and then we just have to check if the start symbol is in $A[0,n]$.

A similar construction is employed in Vijayashanker-Joshi's algorithm [19] too. Vijayashanker-Joshi's algorithm [19] is more complicated than the CKY algorithm since a TAG permits the operation of adjunction (which is not in a CFG). A four dimensional array A is used for TAL parsing.

We assume that the TAG is in normal form, i.e., each node has at most two children. The array A is defined as follows: A node named α is in $A[i,j,k,l]$ if and only if there is a derived tree rooted at α whose *frontier* is given by $a_{i+1}a_{i+2} \ldots a_jYa_{k+1}a_{k+2} \ldots a_l$ where Y is a foot node. The frontier of any tree is defined to be the sequence of labels in the leaves of the tree from left to right.

To begin with $A[i,i+1,i+1,i+1]$ consists of the nodes in the frontier of elementary trees whose label is a_{i+1}, for $0 \leq i \leq (n-1)$. For all $i \leq j$, $A[i,i,j,j]$ contains the foot nodes of all the auxiliary trees.

The algorithm runs in n^4 **phases**, where each phase consists of the following five cases:

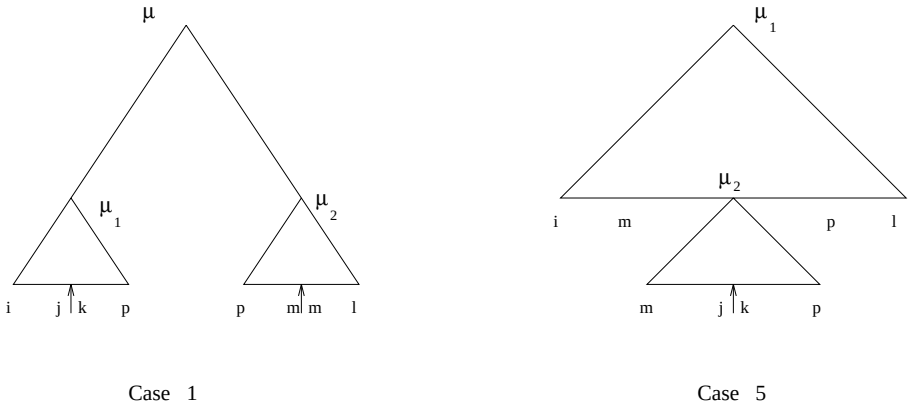

FIG. 3. *Vijayashanker-Joshi's Algorithm*

1. If a node μ_1 is in $A[i,j,k,p]$ and another node μ_2 is in $A[p,m,m,l]$ (for some $k \leq p \leq m$ and $p \leq m \leq l$) such that μ_1 and μ_2 are the left and right children, respectively, of node μ, then the node μ belongs to $A[i,j,k,l]$ if μ_1 is the ancestor of the foot node (see Figure 3).
2. This step is symmetric to step 1. Here there are two nodes μ_1 and μ_2 such that $\mu_1 \in A[i,m,m,p]$, $\mu_2 \in A[p,j,k,l]$ (for some $i \leq m \leq p$ and $m \leq p \leq j$), and these two nodes are the left and right children of node μ. In this case, node μ will belong to $A[i,j,k,l]$.
3. If μ is the parent of $\mu_1 \in A[i,j,j,k]$ and $\mu_2 \in A[k,m,m,l]$, then μ belongs to $A[i,j,j,l]$.
4. If μ's only child is $\mu_1 \in A[i,j,k,l]$, then μ also is in $A[i,j,k,l]$.
5. If node μ_2 is in $A[m,j,k,p]$ and the root $\mu_1 \in A[i,m,p,l]$ of some derived tree α has the same label as that of μ_2, then we could adjoin α at μ_2. Thus, μ_1 belongs to $A[i,j,k,l]$ (see Figure 3).

Clearly, each phase of the algorithm takes $O(n^2)$ time and there are n^4 entries to fill in the array A. Thus the whole algorithm runs in $O(n^6)$ time. Finally, the given input will be in the language if and only if some initial tree is in $A[0,j,j,n]$, $0 \leq j \leq n$. If the grammar size is also taken into account, this algorithm runs in time $O(n^6\ |G|^2)$ with space complexity $O(n^4\ |G|)$. However, the time bound can be reduced to $O(n^6\ |G| \log |G|)$.

4. The New Algorithm. In this section we present an algorithm for CFL parsing that runs in $O(n^3)$ time with a better average run time. Then we extend this algorithm to TAL parsing.

4.1. CFL Parsing. There are two well known algorithms for CFL parsing, namely the CKY algorithm and Earley's algorithm, both of which run in time $O(n^3)$. Furthermore, CFL parsing can be reduced to Boolean matrix multiplication [18]. In this subsection we present an $O(n^3)$ time algorithm that is slightly different from the CKY algorithm. A special feature of this algorithm is that it can be adopted (with some crucial modifications) to parse TALs.

Consider a CFG $G = (N,T,P,S)$ in Chomsky normal form. Each production in P is of the form: $A \rightarrow BC$ or $A \rightarrow a$, where A,B,C are nonterminals and a is a terminal symbol. The basic idea behind the algorithm is the following: The algorithm runs in stages, where in each stage we scan through each production in P and grow

larger and larger parse trees. In particular, at any time, each nonterminal has a list of tuples of the form: (i, j). If A is any nonterminal, $LIST(A)$ will have tuples (i, j) such that A derives $a_{i+1}a_{i+2}\ldots a_j$.

If $A \to BC$ is a production in P, then in any stage we process this production as follows: We scan through elements in $LIST(B)$ and look for matches in $LIST(C)$. For example, if (i, k) is in $LIST(B)$, we check if (k, j) is in $LIST(C)$, for some j. If so, we insert (i, j) into $LIST(A)$ (if it is not already there). Processing of a single production can be done in $O(n^3)$ time, if we maintain the following data structures: 1) For each nonterminal A, an array (call it X_A) of lists indexed 1 through n, where $X_A[i]$ is the list of all tuples from $LIST(A)$ whose first item is i $(1 \leq i \leq n)$; and 2) An $n \times n$ matrix M whose (i, j)th entry will be those nonterminals that derive $a_{i+1}a_{i+2}\ldots a_j$. There can be $O(n^2)$ entries in $LIST(B)$ and for each entry (i, k) in this list, we need to search for at most n items in $LIST(C)$. Thus the total time needed to process a production is $O(n^3)$.

By induction we can show that at the end of stage ℓ $(1 \leq \ell \leq n)$, the algorithm would have computed all the nonterminals that span any input segment of length ℓ or less. (We say a nonterminal spans the input segment $I = a_{i+1}a_{i+2}\ldots a_j$ if it derives I; the nonterminal is said to have a 'span-length' of $j - i$.) Therefore, the algorithm terminates after n stages, implying that the total run time is $O(n^4)$.

However, we can reduce the run time of each stage to $O(n^2)$ as follows: In stage ℓ, while processing the production $A \to BC$, work only with tuples from $LIST(B)$ and $LIST(C)$ whose combination will derive an input segment of length exactly ℓ. For example, if (i, k) is a tuple in $LIST(B)$, the only tuple in C we should look for is $(k, i + \ell)$. We can look for such a tuple in $O(1)$ time using the matrix M. With this modification, each stage of the above algorithm will only take $O(n^2)$ time, yielding the following

LEMMA 4.1. *This algorithm for CFL parsing runs in time $O(n^3)$, with space complexity $O(n^2)$. The algorithm can also construct parse trees while recognizing CFLs.*

Note: This algorithm has an average run time different from the worst case run time. For example, if m is the number of elements that will ever be stored in the matrix M, then, the run time of the above algorithm is no more than $O(mn)$. In fact, this bound can further be tightened using the fact that a rule of the form $A \to BC$, can be processed in time $\min\{|LIST(B)|, |LIST(C)|\}$ by keeping track of the length of each list.

On the other hand, for the CKY algorithm the worst case and average case run times are the same. It is well known that Earley's algorithm also has a different average and worst case run times. Perhaps the performance of our variant of the CKY algorithm compares favorably to that of the CKY algorithm. Also, our algorithm has a run time that grows quadratically with the size of the grammar G, just like the CKY algorithm. However, the run time can be reduced to $O(n^3|G|\log|G|)$ by maintaining each entry of the $n \times n$ array as a red-black tree (or as any other balanced binary tree). The $n \times n$ array itself can be replaced with a Red-Black tree to save space but the time bound will increase by a logarithmic factor. A comparison between Earley's algorithm and ours will be interesting to investigate. A crucial difference between the two is that our algorithm is bottom-up and Earley's is top-down. For a description of Earley's algorithm, see [1].

In related works, Valiant [18] has shown that CFL parsing can be reduced to Boolean matrix multiplication. Similar work for CFL parsing has also been reported

by Graham, Harrison, and Ruzzo [4]. For an excellent treatise on CFL parsing see [1].

4.2. Extension to TALs. In this subsection we show how to extend the above algorithm to parsing TALs. The first algorithm we present will have a run time of $O(n^7)$. In subsection 4.5, we will show how to reduce the run time to $O(n^6)$. We keep track of four indices i, j, k, l corresponding to two different input segments that any derivation tree might span, with a foot node separating these segments, just like in Vijayashanker-Joshi's algorithm [19].

We will adopt the algorithm of the previous section. However, the modified algorithm is complicated by the presence of the adjoin operation. The following assumptions (which have been made in all the existing TAL parsing algorithms) are in place: 1) Each node in any tree has ≤ 2 children; 2) Each auxiliary tree has at least one terminal symbol in its frontier; (However, this assumption can be relaxed). Like in the CFL parsing algorithm, here also we associate a list with each node in each elementary tree. For any node α, $LIST(\alpha)$ at any time will contain quadruples of the form (i, j, k, l) such that the node spans $a_{i+1} \ldots a_j$ and $a_{k+1} \ldots a_l$ with a nonterminal in between. One of the basic operations that the algorithm performs is **Evaluate-Node**. This procedure takes as input a tree node, say γ, and computes $LIST(\gamma)$, given the $LIST$s of its children. This procedure composes the two $LIST$s associated with γ's children.

Evaluate-Node(γ)
/* γ is any node in elementary trees. This procedure computes
$LIST(\gamma)$, given the $LIST$s of γ's children. */
1 **if** γ has no children **then**
2 do nothing
3 **if** γ has one child, say α, **then**
4 $LIST(\gamma) := LIST(\alpha)$
5 **if** γ has two children, say α and β, **then**
6 **for every** $(i, j, k, l) \in LIST(\alpha)$ **do**
7 **for every** $l \leq m \leq p \leq n$ **do**
8 **if** $(l, m, m, p) \in LIST(\beta)$ and $(i, j, k, p) \notin LIST(\gamma)$ **then**
9 insert (i, j, k, p) into $LIST(\gamma)$
10 **end for**
11 **end for**
12 **for every** $(l, m, p, q) \in LIST(\beta)$ **do**
13 **for every** $1 \leq i \leq j \leq l$ **do**
14 **if** $(i, j, j, l) \in LIST(\alpha)$ and $(i, m, p, q) \notin LIST(\gamma)$ **then**
15 insert (i, m, p, q) into $LIST(\gamma)$
16 **end for**
17 **end for**

Here also we maintain an $n^2 \times n^2$ array, say Γ. $\Gamma[i, j, k, l]$ will store all the nodes in elementary trees that are known to span the input segments $a_{i+1} \ldots a_j$ and $a_{k+1} \ldots a_l$, $1 \leq i, j, k, l \leq n$. In this paper $1 \leq i, j, k, l \leq n$ is short hand for $1 \leq i \leq j \leq k \leq l \leq n$. With such a data structure, we could perform **Evaluate-Node**(γ), for any node γ in time $O(n^6)$, since there can be no more than n^4 elements in the $LIST$ of any node. Lines 1 through 4 of **Evaluate-Node** correspond to Step 4 of Vijayashanker-Joshi's algorithm. The rest of the lines in **Evaluate-Node** correspond to Steps 1 through 3 of their algorithm.

We also employ a procedure called **Evaluate-Tree** which works in a bottom-up fashion evaluating nodes at each level. Basically, this procedure updates the $LIST$ of every node. If γ is an adjoin node labelled X, we also perform **Adjoin**(γ, X).

The procedure **Adjoin**(α, X) performs adjunction at the node α, where X is the label of the node α. **Adjoin** corresponds to Step 5 of Vijayashanker-Joshi's algorithm.

Finally, we have the main procedure called **TAL-Parse** where we evaluate each auxiliary tree and then we evaluate each initial tree.

Evaluate-Tree(T)
/* This procedure '**Evaluates**' all the nodes of tree T level by level
starting from the bottom-most level */
1 **for** each node γ *in* T **do**
2 **Evaluate-Node**(γ)
3 **if** γ is an 'adjoin' node labelled X **then**
4 **Adjoin**(γ, X)
5 **end for**

Adjoin$(\alpha,\ X)$
/* This procedure performs the adjoin operation at node α that is labelled by
the nonterminal X; R is an array of n^2 lists.
β is an auxiliary tree labelled by X. */
1 Scan through $LIST(\alpha)$ and for each quadruple of the form
(q, j, k, r) encountered, add (j, k) to the list $R[q, r]$
2 **for** each $(i, q, r, l) \in LIST(\beta)$ **do**
for each $(j, k) \in R[q, r]$ **do**
Put (i, j, k, l) into $LIST(\alpha)$ if it is not there

TAL-Parse
/* This algorithm takes as input a TAG G and an input $a_1 a_2 \ldots a_n$ and
returns 'YES' if the input string is in $L(G)$ and 'NO' otherwise */
0 Initialize the array Γ and $LIST(\gamma)$ for each node γ in elementary trees
1 **for** $\ell\ :=\ 1$ **to** n **do**
2 **for** each auxiliary tree γ **do**
3 **Evaluate-Tree**(γ)
4 **end for**
5 **for** each initial tree γ **do**
6 **Evaluate-Tree**(γ)
7 **end for**
8 **end for**
9 **if** the root of an initial tree $\in\ \Gamma[0, j, j, n]$ for some $0 \leq j \leq n$ **then**
10 **return** YES **else return** NO

Note: The need for performing **Evaluate-Tree** first on the auxiliary trees will become clear in the correctness proof given in subsection 4.4. This is done to account for initial trees that may not have any non-ϵ terminal symbols in their frontiers. For example, there could be an initial tree of the form: $S\ ---\ S\ ---\ \epsilon$, where the second S node is an adjunction node. Note also that the data structure Γ is crucial to the algorithm. It enables us to perform the following operations efficiently: 1) Update $LIST$s of nodes; 2) Check if a given 4-tuple is in $LIST(\alpha)$ for some given α; etc.

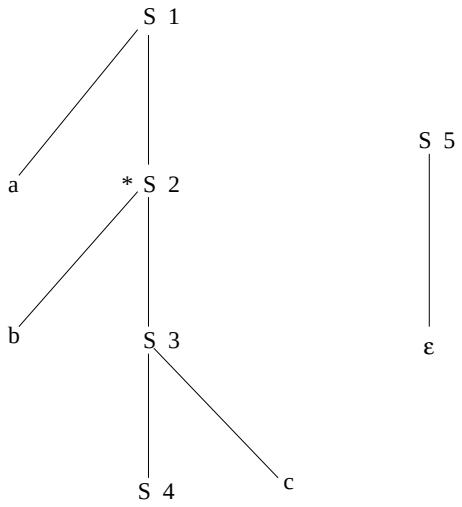

FIG. 4. *A TAG for* $\{a^n b^n c^n | n \geq 0\}$.

4.3. An Example. Before proving the correctness of **TAL-Parse**, we demonstrate the above basic algorithm with an example. Consider the following language $\mathcal{L}$: $\{a^n b^n c^n | n \geq 0\}$. Figure 4 shows a TAG that generates $\mathcal{L}$. Here $*$ denotes the adjoin node. Integers ranging from 1 to 5 are used to label nodes and the label of each node is shown next to it. The input considered is *aabbcc*.
Stage 1: $LIST(4)$ is $\emptyset$. $LIST(3)$ is computed as $(4, i, i, 5)$ for every i and $(5, j, j, 6)$ for every j. $LIST(2) = (2, 3, 4, 5)$, $(3, 4, 4, 5)$, $(2, 3, 5, 6)$, $(3, 4, 5, 6)$. Finally, $LIST(1)$ is computed as $(0, 1, 3, 5)$, $(0, 2, 3, 5)$, $(1, 3, 4, 5)$, $(1, 3, 5, 6)$.
Stage 2: $LIST(4)$ still remains empty. $LIST(3)$ also remains the same. $LIST(2)$ gets modified because of the adjunction performed at this node. $LIST(2)$ gets a new element: $(0, 4, 4, 6)$. As a result, $LIST(1)$ also gets a new member: $(0, 4, 4, 6)$. In this case **TAL-Parse** returns 'YES'.

4.4. Correctness of TAL-Parse. In this subsection we prove the following Lemma.

LEMMA 4.2. *The algorithm* **TAL-Parse** *is correct.*

Proof. The proof will be by induction on ℓ. The execution of instructions 2 through 7 will constitute a 'stage' of the algorithm. In the following proof, by a 'terminal symbol' we mean a terminal symbol other than ϵ.

Induction Hypothesis: After the ℓth stage of the algorithm, all tree nodes that span an input segment of length ℓ or less will have been identified. We say a node spans a string of length ℓ if it spans $a_{i+1} \ldots a_j$ and $a_{k+1} \ldots a_m$ such that sum of lengths of these two segments is ℓ.

Base case ($\ell = 1$): If γ is any node that spans an input segment of length 1, then there are three possibilities: 1) γ is labelled by a terminal symbol; 2) the frontier of the subtree rooted at γ has a terminal symbol; or 3) γ has neither a terminal for a label nor a terminal in its frontier. Case 1 is trivial. In case 2, **Evaluate-Tree** will surely infer that γ spans an input of length 1. In case 3, γ can span an input segment of length 1 only with the help of an adjoin operation. In particular, the auxiliary tree that is adjoined should have a span of length 1. But the roots of all the auxiliary

trees fall under case 2 and hence would have been correctly processed in lines 2 and 3 of **TAL-Parse**.

The induction step. Assume the hypothesis for all stages $\leq \ell$. We'll prove it for the $(\ell+1)$th stage.

Let γ be any tree node that spans an input segment of length $\ell+1$. The corresponding derivation can be obtained with a sequence of basic operations, namely, composition and adjoin performed on trees whose span-lengths are ℓ or less. Some of these operations might not have added to the span-length of the derived tree. Call these operations useless and others useful. Let α be the node in this tree at which the last (from bottom-up) useful operation was performed. Now there are two cases to consider.

Case 1: The last useful operation performed was a composition, say of two subtrees (of span-length at least one each). Clearly both of these subtrees have a span-length of ℓ or less and hence, using the induction hypothesis, would have already been identified. Therefore, the procedure **Evaluate-Tree** when performed on γ will identify the fact that γ spans an input segment of length $\ell+1$.

Case 2: The last useful operation performed was adjoin, say at a node α labelled X. This means that there is a quadruple, say (i,j,k,m), in $LIST(\alpha)$ such that when a derived auxiliary tree labelled X is adjoined at α, the resultant tree has a span-length of $\ell+1$. Let δ be the derived tree corresponding to the quadruple (i,j,k,m) and θ be the auxiliary tree that is adjoined. In this case, if δ has at least one terminal in its frontier, then θ will have a span-length of at most ℓ. In θ each constituent subtree will have a span-length of ℓ or less and hence would have been already identified. Thus, the algorithm **Adjoin** will identify α as having a span-length of $\ell+1$.

On the other hand if δ does not have any terminal in its frontier, then, θ will have a span-length of $\ell+1$. But auxiliary trees have at least one terminal in their frontiers and hence would have been processed (by **TAL-Parse**) in lines 2 and 3 of stage $\ell+1$. Therefore, in this case also, the procedure **Adjoin** will identify α as having a span-length of $\ell+1$. □

Time and Space Analysis. Since $LIST(\gamma)$ will have $O(n^4)$ quadruples for any node γ, **Evaluate-Node**(γ) takes time $O(n^6)$. In fact, **Evaluate-Node** can be run in $O(n^5)$ time. We only mention how to modify steps 6 through 11: (R is an array of lists indexed from 1 to n).

```
1   Scan through LIST(β) and for each quadruple of the form
2   (l, m, m, p), add p to the list R[l]
3   for each (i, j, k, l) in LIST(α) do
4       for each p ∈ R[l] do
5           Put (i, j, k, p) into LIST(γ) if it is not there
```

Adjoin takes time $O(n^6)$. Thus, for any tree T, **Evaluate-Tree**(T) also runs in time $O(n^6)$, implying that the run time of **TAL-Parse** is $O(n^7)$. Clearly, the space used is $O(n^4)$. As a result, we get the following:

THEOREM 4.3. *The algorithm* **TAL-Parse** *takes* $O(n^7)$ *time and* $O(n^4)$ *space.*

4.5. Reducing the Run Time to $O(n^6)$. In subsection 4.1 we reduced the time cost of our variant of the CKY CFL parser from $O(n^4)$ down to $O(n^3)$. In this subsection we use the same technique for TALs also. That is, in stage q, we only generate trees of span-length exactly q.

We can process steps 6 through 11 of **Evaluate-Node** in $O(n^4)$ time as follows:

Let R be an $n \times n$ matrix initialized to all zeros and Q be a list of tuples. In stage q of **TAL-Parse** ($1 \leq q \leq n$), we only generate quadruples whose total span length is exactly q.

1 Scan through $LIST(\beta)$ and for each quadruple of the form
2 (l, m, m, p) encountered, mark $R[l, p]$ and add (l, p) to Q
3 **for** each (i, j, k, l) in $LIST(\alpha)$ **do**
4 Let $p = l + q - [(j - i) + (l - k)]$.
5 **if** $R[l, p]$ is marked **then**
6 Put (i, j, k, p) into $LIST(\gamma)$ if it is not there
7 Using the list Q, clean the matrix R for future use

Similarly, we could process **Adjoin** in $O(n^5)$ time as follows: (Here α is the node into which the auxiliary tree rooted at β is adjoined. R is an $n^2 \times n^2$ matrix initialized to zeros.)

1 Scan through $LIST(\alpha)$ and for each quadruple of the form
2 (m, j, k, r) encountered, mark $R[m, j, k, r]$
3 **for** each (i, m, r, l) in $LIST(\beta)$ **do**
4 Let $p = q - [(m - i) + (l - r)]$.
5 **for** $v := 0$ **to** p **do**
6 **if** $R[m, m + v, r - p + v, r]$ is marked **then**
7 Add $(i, m + v, r - p + v, l)$ into $LIST(\beta)$ if it is not there
8 Using the list β, clean the matrix R for future use

Thus it becomes clear that **Evaluate-Tree** runs in time $O(n^5)$ yielding the following

THEOREM 4.4. *The modified version of* **TAL-Parse** *runs in time* $O(n^6)$ *using* $O(n^4)$ *space.*

Note: The worst case and average case run times of our algorithm are different. We believe that this algorithm will perform well in practice.

5. A Faster Algorithm for TAL Parsing. In this section we show that TAL parsing can be done in time $O(n\ M(n^2))$, where $M(k)$ is the time needed for multiplying two Boolean matrices of size $k \times k$ each. We make use of the algorithm of section 4.2.

The claim follows from the following Lemmas:

LEMMA 5.1. **Evaluate-Node**(γ) *can be processed in time* $n^2\ M(n)$ *for any node* γ.

Proof: It suffices to show how we could process lines 6 through 11 of **Evaluate-Node**(γ) within the stated time bound. Construct two Boolean matrices A and B corresponding to elements in $LIST(\alpha)$ and $LIST(\beta)$ as follows: The matrices will be of size $n^2 \times n^2$ each. Rows of the matrices are named with tuples of the form (i, j) and columns are numbered with tuples of the form (k, l) ($1 \leq i, j, k, l \leq n$). $A[i, j; k, l]$ will be 1 if and only if (i, j, k, l) is a quadruple in $LIST(\alpha)$. Similarly define B.

Clearly, (i, j, k, p) will be in $LIST(\gamma)$ if and only if

$$f_{ijkp} = \sum_{l=1}^{n} \sum_{m=1}^{n} A[i, j; k, l]\ B[l, m; m, p]$$

is a 1. But,

$$\sum_{l=1}^{n} \sum_{m=1}^{n} A[i, j; k, l]\ B[l, m; m, p] = \sum_{l} A[i, j; k, l]\ (\sum_{m} B[l, m; m, p]).$$

The above fact suggests the following strategy for processing **Evaluate-Node**(γ): Let V_{ik} be the $n \times n$ sub-matrix of A defined as follows.

$$V_{ik} \;=\; \{A[i,j;k,l]\}\,,\; 1 \leq j,l \leq n,$$

and let C be an $n \times n$ matrix such that $C[p,l] \;=\; \sum_{q=1}^{n}\; B[p,q;q,l]$, $1 \leq p,l \leq n$. Now, f_{ijkp} is nothing but the dot product of the jth row of V_{ik} and the pth column of C. Of course the jth row in V_{ik} is nothing but all the elements $A[i,j;k,l]$, $1 \leq l \leq n$.

Therefore, multiplying the two $n \times n$ matrices V_{ik} and C yields the values of f_{ijkp} for $1 \leq j,p \leq n$. Hence, obtaining values of all possible f_{ijkp}'s takes n^2 such multiplications. □

LEMMA 5.2. **Adjoin** *can be performed in time* $M(n^2)$.

Proof. Say we have to adjoin at a node labelled α. Let the auxiliary tree to be adjoined be β. We have to generate all quadruples of the form (i,j,k,l) such that (i,q,r,l) is in $LIST(\beta)$ and (q,j,k,r) is in $LIST(\alpha)$. This can be done as follows. Generate matrices U and W for $LIST(\alpha)$ and $LIST(\beta)$, respectively, as above. Now the array U can be indexed (i.e., permuted) in such a way that the ilth row of U has the following elements:

$$U[i,1;1,l], U[i,1;2,l], \ldots, U[i,1;n,l],\; U[i,2;1,l], U[i,2;2,l], \ldots, U[i,2;n,l],$$

$$\ldots,\; U[i,n;n,l].$$

Likewise index (i.e., permute) the elements of W such that the jkth column has the elements:

$$W[1,j;k,1], W[1,j;k,2], \ldots, W[1,j;k,n],\; W[2,j;k,1], W[2,j;k,2], \ldots, W[2,j;k,n],$$

$$\ldots,\; W[n,j;k,n].$$

Clearly, the dot product and then Boolean OR of the ilth row of U and jkth column of W will be a 1 if and only if the adjoin results in the quadruple (i,j,k,l). □

As a result we get (making use of the algorithm in Section 4.2 and the above Lemmas):

THEOREM 5.3. *The algorithm* **TAL-Parse** *costs* $O(n\; M(n^2))$ *time and* $O(n^4)$ *space.*

Note: The above algorithm has a run time of $O(n^{5.752})$ which already breaks the $\Theta(n^6)$ barrier.

6. An $O(n^3\; M(n))$ Time Algorithm. We now show that TAL parsing can be done in time $O(n^3\; M(n))$. Since the best known value for $M(k)$ is $O(k^{2.376})$, the time bound of our algorithm is $O(n^{5.376})$. The key idea is to use the algorithm given in Section 4.2 together with asymptotically fast algorithms for Boolean matrix multiplication. In stage q of the algorithm we only process quadruples whose span length is exactly equal to q, $1 \leq q \leq n$. Boolean matrix multiplication has been previously employed in the design of parsing algorithms [18] [4]as has been mentioned before.

It should be noted here that it suffices to show how **Adjoin** can be performed at any node in $O(n^2\; M(n))$ time in any given stage of the algorithm.

There can be only $O(n^3)$ quadruples of the form (i,j,k,l) whose span length is exactly equal to q (for any q). These quadruples can be classified according to the value of $k-j$ as follows: Call a quadruple 'a class m quadruple' if $k-j=m$. Clearly, there are at most n such classes and there are at most n^2 quadruples in each class whose span length is q.

Consider any arbitrary class (say m) of quadruples. Let (i',j',k',l') be any quadruple in this class. Then, the other quadruples (i,j,k,l) in this class can be enumerated as follows: $i=i'\pm c;\ l=l'\pm c;\ j=j'\pm d; k=k'\pm d$ for any integers c and d (as long as the tuple (i,j,k,l) is a meaningful one). That is, there are at most n choices for the pair i,l and independently there are at most n choices for the pair j,k.

Now construct two Boolean matrices J and K as follows: The ilth row of J will have entries corresponding to the quadruples

$$(i,1,1,l),(i,1,2,l),\ldots,(i,1,n,l),\ (i,2,1,l),(i,2,2,l),\ldots,(i,2,n,l),\ \ldots,\ (i,n,n,l).$$

Here il ranges over all possible values for the pair i,l indicated above. The entry corresponding to a quadruple (i,j,k,l) will be 1 if the node under consideration is known to span it. Similarly, the jkth column of K will have entries corresponding to the quadruples:

$$(1,j,k,1),(1,j,k,2),\ldots,(1,j,k,n),\ (2,j,k,1),(2,j,k,2),\ldots,(2,j,k,n),\ \ldots,\ (n,j,k,n).$$

Here jk ranges over all possible values for the pair j,k indicated above.

Now, the product of the two matrices J and K will give information about all new quadruples in class m that arise out of adjunction at the node under concern. That is, a quadruple (i,j,k,l) in class m is spanned by the node under consideration only if the corresponding entry in the product is 1. Since J and K are matrices of size $n\times n^2$ and $n^2\times n$, respectively, they can be multiplied in time $O(n\ M(n))$. Therefore, we can compute all the quadruples belonging to a specific class and spanned by the node under consideration in $O(n\ M(n))$ time. The technique employed for multiplication of J and K is as follows: Partition the matrix J into n submatrices $J_1,J_2,\ldots,J_n$ where J_i consists of the ith leftmost block of n columns of J. Similarly partition K into $K_1,K_2,\ldots,K_n$ where K_i is the ith topmost block of n rows of K, $1\le i\le n$. Now, clearly, the block-wise product of J and K is the same as $\sum_{i=1}^{n} J_i\ K_i$. This implies that we can obtain all the quadruples spanned by this node (of a specific span length) in time $O(n^2\ M(n))$.

As a result, we have shown the following.

THEOREM 6.1. *TAL parsing can be done in time $O(n^3\ M(n))$ with space complexity $O(n^4)$.*

The following theorem pertains to parallel implementation of the above ideas. In **TAL-Parse**, we can run everything in parallel fine, except we process one span length at a time.

THEOREM 6.2. *TAL parsing can be done on the EREW PRAM in $O(n\log n)$ time using $\frac{n^2\ M(n)}{\log n}$ processors.*

Proof. The Theorem can be proven using the fact that two $n\times n$ matrices can be multiplied in $O(\log n)$ time on an EREW PRAM using $\frac{M(n)}{\log n}$ processors [7]. If the above algorithms are analyzed using this fact, the Theorem readily follows. The PT bound of this algorithm is asymptotically the same as the sequential run time. Therefore, our parallel algorithm is an optimal implementation of the sequential algorithm. □

7. Conclusions. We have resolved the open question as to whether TAL parsing can be done in time $o(n^6)$. Our algorithm also runs efficiently in parallel. We have presented algorithms for CFL and TAL parsing which may perform well in practice. Our parallel algorithm has time cost asymptotically within a log factor of any prior parallel algorithm for TAL parsing. At the same time, our algorithm uses significantly fewer processors. It is still unknown whether there is a *practical* algorithm for TAL parsing that runs in time $o(n^6)$.

Acknowledgements. I am grateful to the referees for their critical comments and suggestions. I am thankful to Günter Hotz and Aravind K. Joshi for their support and encouragement. I am also thankful to B. Srinivas and Shibu Yooseph for many stimulating discussions.

REFERENCES

[1] A. Aho and J. D. Ullman, *The theory of parsing, translating, and compiling*, Volume 1, Addison-Wesley Publishing Company, 1984.

[2] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, Journal of Symbolic Computation, Vol. 9, 1990, pp. 251-280. Also in Proc. 19th Annual ACM Symposium on Theory of Computing, 1987, pp. 1-6.

[3] G. Gazdar, *Applicability of indexed grammars to natural languages*, Technical Report CSLI-85-34, Center for Study of Language and Information, 1985.

[4] S. L. Graham, M. A. Harrison, and W. L. Ruzzo, *On line context free language recognition in less than cubic time*, Proc. 8th Annual ACM Symposium on Theory of Computing, 1976, pp. 112-120.

[5] Y. Guan and G. Hotz, *An $O(n^5)$ recognition algorithm for coupled parenthesis rewriting systems*, in Proc. TAG+ Workshop, University of Pennsylvania, Philadelphia, June 1992.

[6] K. Harbusch, *An efficient parsing algorithm for tree adjoining grammars*, in Proc. 28th Meeting of the Association for Computational Linguistics, Pittsburgh, 1990, pp. 284-291.

[7] J. JÁ JÁ, *An Introduction to Parallel Algorithms*, Addison-Wesley Publishing Company, 1992, pp. 248-248.

[8] A. K. Joshi, *How much context-sensitivity is necessary for characterizing structural descriptions – tree adjoining grammars*, in *Natural Language Processing – Theoretical, Computational and Psychological Perspective*, edited by D. Dowty, L. Karttunen, and A. Zwicky, Cambridge University Press, New York, NY 1985.

[9] A. K. Joshi, L. S. Levy, and M. Takahashi, *Tree adjunct grammars*, Journal of Computer and System Sciences, 10(1), 1975, pp. 136-163.

[10] A. Kroch and A. K. Joshi, *Linguistic relevance of tree adjoining grammars*, Technical Report MS-CIS-85-18, Department of Computer and Information Science, University of Pennsylvania, 1985.

[11] T. Nurkkala and V. Kumar, *A parallel parsing algorithm for natural language using tree adjoining grammar*, Proc. 8th International Parallel Processing Symposium, 1994.

[12] M. Palis, S. Shende, and D. S. L. Wei, *An optimal linear time parallel parser for tree adjoining languages*, SIAM Journal on Computing, 19(1), 1990, pp. 1-31.

[13] B. H. Partee, A. Ter Meulen, and R. E. Wall, *Studies in Linguistics and Philosophy*, Vol. 30, Kluwer Academic Publishers, 1990.

[14] G. Satta, *Tree adjoining grammar parsing and Boolean matrix multiplication*, Proc. 32nd Meeting of the Association for Computational Linguistics, 1994.

[15] G. Satta, Personal Communication, September 1993.

[16] Y. Schabes and A. K. Joshi, *An Earley-type parsing algorithm for tree adjoining grammars*, Proc. 26th Meeting of the Association for Computational Linguistics, 1988, pp. 258-269.

[17] M. Steedman, *Dependency and coordination in the grammar of Dutch and English*, Language, 61, 523-568, 1985.

[18] L. G. Valiant, *General context-free recognition in less than cubic time*, Journal of Computer and System Sciences, 10, 1975, pp. 308-315.

[19] K. Vijayashanker and A. K. Joshi, *Some computational properties of tree adjoining grammars*, Proc. 23rd Meeting of the Association for Computational Linguistics, 1985, pp. 82-93.