INFORMATION AND COMPUTATION 101, 33–69 (1992)

Tight Complexity Bounds for Term Matching Problems*

RAKESH M. VERMA†

Department of Computer Science, University of Houston, Houston, Texas 77204

AND

I. V. RAMAKRISHNAN‡

Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794

In this paper we study the sequential and parallel complexity of various important term matching problems. These problems occur frequently in applications such as term rewriting, functional programming, and logic programming. First, we obtain non-trivial lower bounds on the parallel complexity of the (uninterpreted) term matching problem. We also establish the tightness of these bounds for some representations and several models. We then characterize completely the sequential complexity of associative and commutative matching when the number of occurrences of variables is varied. Specifically, we show that even if each variable is restricted to at most two occurrences in the terms, both associative matching and commutative matching are NP-complete. Interestingly, for the important restriction of boolean terms we show that commutative matching is NP-complete whereas associative matching has a linear time sequential algorithm. For linear terms, we significantly improve the existing upper bound for associative–commutative matching, and present a new algorithm for associative matching. Designing direct parallel algorithms for associative–commutative matching of linear terms appears to be a difficult task. Despite this we have been able to resolve questions about the parallel complexity of these problems using complexity-theoretic techniques. Finally, an interesting consequence of the research reported here is the demonstration of a tighter relationship between associative–commutative matching for linear terms and bipartite matching on both sequential and parallel models.

* A preliminary version of some of these results was presented at the 9th Conference on Automated Deduction, May 1988, and the Symposium on Theoretical Aspects of Computer Science, February 1989.

† Research supported in part by NSF under Grant CCR-9010366, and by a University of Houston research initiation grant. A substantial portion of this research was done while the author was at SUNY Stony Brook.

‡ Research supported in part by NSF under Grant CCR-8805734.

0890-5401/92 $5.00

1. INTRODUCTION

Reasoning with equations is important in logic, mathematics, and many computing applications such as algebraic specifications, equational and functional programming languages, and automated deduction. Two popular mechanical approaches for reasoning with equations are resolution augmented with paramodulation and term rewriting. These approaches use unification and matching procedures extensively. Informally, given two terms t_1 and t_2 the unification (decision) problem is to determine whether there exists a substitution σ for the variables in the two terms such that $\sigma(t_1) = \sigma(t_2)$. In terms matching the substitution is applied to only one term which is called the *pattern*; the other term is called the *subject*. Unification and matching are also the most frequently repeated operations in the execution of Prolog programs. Consequently, a lot of attention has been devoted to speeding them up using parallel processing. Several theoretical studies of the parallelism in these operations have been carried out on the Parallel Random Access Machine model (PRAM) (Fortune and Wyllie, 1978) and its variants such as CRCW (concurrent read concurrent write), CREW (concurrent read exclusive write), and Arbitrary PRAM.

Dwork *et al.* (1984) showed that a polylogarithmic time parallel algorithm, using polynomially many processors, is unlikely for unification, whereas term matching is in NC (Cook, 1985). These results, coupled with the importance of term matching, led to a surge of interest in designing efficient parallel algorithms for it (Dwork *et al.*, 1986; Ramesh *et al.*, 1989). However, the following important question remained unanswered: Is there a limit to the parallelism achievable in term matching (of course, the processor requirement should be reasonable). For instance, can it be done in constant parallel time? We show that this is unfortunately not possible.

▶ We show that on any CRCW PRAM, term matching requires $\Omega(\log n/\log\log n)$ time independent of the representation of terms (strings stored in arrays, trees, or directed acyclic graphs) as long as the number of processors is restricted to a polynomial in the input size. We also prove that this lower bound is tight for the string representation, by giving corresponding optimal time upper bounds on the Arbitrary PRAM. Note that on the Arbitrary PRAM string matching can be done in $O(1)$ time. Thus our lower bounds for term matching imply that the presence of variables makes term matching provably more difficult than string matching. We also prove $\Omega(\log n)$ time lower bounds for term matching on the CREW PRAM for all three representations. We then show that these lower bounds are tight for the string and tree representations. All our lower bounds hold for both linear[1] as well as non-linear terms, whereas our

[1] If each variable is restricted to at most one occurrence, then the term is *linear*.

upper bounds are for the general case of non-linear terms. Thus, on the PRAM model, non-linearity is not the inherent source of difficulty in uninterpreted term matching. These results settle some open problems raised in Dwork *et al.* (1984).

In many applications of interest, some function symbols in the terms are interpreted, i.e., they satisfy certain axioms. Some of the most frequently occurring axioms are associativity and commutativity. For example, the *Append* operation in LISP is associative. Logical connectives such as OR and AND in first order logic, and operations in Abelian groups and fields are both associative and commutative. Extensions of equational reasoning methods to handle commutativity and associativity use specialized unification and matching procedures. The widespread occurrence of these two axioms, in practice, makes it imperative to study the computational complexity of these procedures.

In this paper we also study the sequential complexity of associative commutative matching (ACM) and its variants such as associative matching (AM), commutative matching (CM), and their *linear* versions which occur in left-linear rewrite systems. A rich theory of such systems has been developed and utilized in the design of equational programming languages (O'Donnell, 1985). The following is a brief description of our results on ACM and its variants.

▶ Using a result of the first author (Verma, 1988), which combines a subtle modification of the reduction from linear ACM to bipartite matching with tighter analysis, we prove a "tight" upper bound on the time required, $O(|s|\ |t|^{1.5})$, for linear ACM. Thus we significantly improve the time bound of $O(|s|\ |t|^3)$ for linear ACM given in Benanav (1985) ($|s|$ is the size of the pattern and $|t|$ is the size of the subject). Moreover, we give a linear time reduction from bipartite matching to linear ACM. An important consequence of this mutual reducibility is the tightness of our upper bound, since any nontrivial improvement in the running time of one problem will improve that of the other.

▶ For the case of linear AM and CM we are able to do even better. Specifically, we present an $O(|s|\ |t| \log |s|)$ time algorithm for linear AM, and an $O(|s|\ |t|)$ time algorithm for linear CM.

▶ We completely characterize the sequential complexity of ACM when the number of occurrences of variables is varied. Specifically, we show that AC matching is NP-complete even if variables are restricted to at most two occurrences in the pattern. In Benanav (1985) it was claimed that ACM is NP-complete even when each variable is restricted to at most 3-occurrences. Thus the complexity of 2-occurrence ACM was left open.

▶ In several applications of theorem proving the ACM problem along with its variants does not occur in its full generality. In particular we

consider the important restriction of this problem to boolean terms. In such terms the operands of AC operators cannot be variables. Matching boolean terms has special significance in theorem proving. For example, the subsumption test is nothing but boolean matching. Similarly, the AC operators, $\vee$ and $\wedge$, cannot have variables as operands in the well-formed formulas of first-order logic. We show that boolean CM is NP-complete whereas boolean AM can be done in linear time. This result provides powerful justification, from a complexity viewpoint, for the following approach to theorem proving involving boolean terms: incorporate associativity in the matching procedure, and use the commutative axioms in the direction specified by a complete simplification ordering (Bachmair *et al.*, 1989). Note that such an ordering *always* exists. The problem usually is to find one.

Finally, we examine the parallel complexity of AM, CM, and ACM. Obviously we consider only linear terms. Parallel complexity results for these problems were hitherto unknown. Designing (direct) parallel algorithms for these problems on the PRAM model appears to be extremely difficult. One of our contributions in this paper is that, despite this difficulty, we have been able to resolve their parallel complexity using complexity-theoretic techniques. Our approach is particularly well suited for problems that are not amenable to direct parallelization.

▶ We show that linear AM and CM are in Nlogspace $\subseteq NC^2$. Next we show that a restricted version of linear ACM is NC reducible to subtree isomorphism. Finally, for the parallel case also, we establish an interesting relationship between linear ACM and bipartite matching. We show that they are mutually NC reducible. Thus the fate of linear ACM is again tightly linked to that of bipartite matching.

The rest of this paper is organized as follows. In the next section we present the necessary notations and definitions. In Section 3 we describe our parallel complexity results for term matching. Section 4 deals with the complexity of AC matching and its variants. Concluding remarks appear in Section 5.

2. Problem Definition

Let $V = \{v_1, v_2, \ldots\}$ be a countable set of variables and F a countable set of function symbols. Each function symbol $f \in F$ has a fixed arity a_f, a non-negative integer. We require that the set of 0-ary function symbols (also called constants) be non-empty, and that $V \cap F = \varnothing$. The set of *terms* $\mathcal{T}$ is defined as follows:

DEFINITION 1.

- a variable or a constant is a term, and
- if $f \in F$ and $t_1, ..., t_{a_f}$ are terms, then so is $f(t_1, ..., t_{a_f})$.

A term can be represented in three different ways: as a labeled directed acyclic graph (DAG), as a fully parenthesized string stored in an array, or by a *labeled directed tree* defined in the obvious way. The DAGs and trees are ordered. The *size* of a term is the number of symbols in it, omitting parentheses and commas. This is the same as the sizes of the tree and string representations of the term, but potentially larger than the size of the DAG (number of vertices + number of edges) representing it.

The tree and DAG representations of terms are very popular. In parallel computation the choice of representation can make a dramatic difference in the complexity of the problem. For example, the circuit value problem (here the circuit is a DAG) is log space complete for P, whereas if the circuit is a tree the same problem is easily seen to be in NC. Therefore, we examine a third representation, the string representation, to see whether there is a "natural" representation for which the parallel complexity of term matching is significantly different. The string representation has also gained much attention recently for sequential symbolic algorithms. Several fast sequential algorithms for various problems involving terms have been reported by Ramesh *et al.* (1990) and Christian (1989). In all these algorithms terms were represented as strings.

DEFINITION 2. A substitution σ is a function from V to $\mathcal{T}$ such that $\sigma(v) \neq v$ for only finitely many variables v.

Notation. 1. For any positive integer k, I_k denotes the set $\{1, 2, ..., k\}$. I_0 is the empty set $\varnothing$.

2. The size of a term t, the cardinality of a set S, and the number of elements in a tuple o are denoted $|t|$, $|S|$, and $|o|$, respectively.

3. Unless stated otherwise σ and its subscripted versions denote substitutions.

4. The application of a substitution σ to a term t is written $\sigma(t)$.

5. V_s denotes the set of variables occurring in a term s.

6. The restriction of a substitution σ to $V' \subseteq V$ is written $\sigma|_{V'}$.

7. If s is any rooted tree and v any vertex in s then Δ_v^s denotes the subtree of s rooted at v.

Remark 1. We assume that the reader is familiar with the notion of a polynomial-time many-one reduction ($\leqslant_m^P$) (Garey and Johnson, 1979), an

NC reduction ($\leqslant_{NC}$) (Cook, 1985), and the complexity classes NL (Nlogspace) and NC.

Throughout this paper s will represent the pattern and t the subject. An *equation* is a pair of terms s, t, written $s = t$ (variables in s and t are implicitly universally quantified).

DEFINITION 3. Let E be any set of equations. Two terms $s, t \in \mathcal{T}$ are E equivalent, written $s =_E t$, iff $\text{Th}(E) \models s = t$, where $\text{Th}(E)$ is the equational theory axiomatized by E. We say that s E-matches t iff there exists a σ such that $\sigma(s) =_E t$.

When E consists of associativity axioms only, we have the AM problem: Given two terms s and t does s match t modulo associativity? Similarly, when E consists of commutativity axioms only, we have the CM problem. When E consists of both associativity and commutativity axioms we have the ACM problem, and when $E = \varnothing$ we have the term matching problem defined above. By abuse of notation, we write $\sigma(s) =_A t$ and say that s a-matches t when s matches t modulo some associativity axioms, and similarly $\sigma(s) =_C t$, etc. In ACM, E must consist of both associative and commutative axioms for every interpreted operator. To elaborate, in ACM the input terms are constructed out of a non-empty set of AC operators and a non-empty set of free (uninterpreted) operators. The most general AC matching problem, in which terms can have some free operators, some operators which are associative only, some which are commutative only, and some AC operators, will be denoted by GACM.

When the set of terms is restricted to boolean terms (i.e., no interpreted operator can have a variable as an argument) we have the boolean versions of these problems. Similarly, when terms are required to be linear we have the linear versions of these problems (denoted by AML, CML, ACML, and GACML). Finally, when terms can contain at most two occurrences of each variable, we have the 2-occurrence AM, 2-occurrence CM, 2-occurrence ACM, and 2-occurrence GACM problems.

3. PARALLEL TERM MATCHING

We begin with a brief description of the model. The PRAM model consists of a collection of processors—each processor has a unique integer ID that it knows—which share a global memory and execute a single program in lock step. There are variants of the PRAM model which handle concurrent reads and writes to the same global memory location differently. The strongest variant is the CRCW PRAM (Concurrent Read

Concurrent Write) which allows simultaneous reads and simultaneous writes to the same location. The CREW PRAM (Concurrent Read Exclusive Write) allows simultaneous reads of a memory location, but not simultaneous writes. The weakest variant is the EREW PRAM (Exclusive Read Exclusive Write) in which, at any given time, at most one processor can access any memory location. Three different variants of the CRCW PRAM have been proposed in the literature. The first and the weakest is the Common PRAM in which all processors attempting to write in the same location simultaneously must write the same value. The second is the Arbitrary PRAM in which one of the competing processors succeeds in writing, but no assumption can be made as to which processor succeeds. The third and the strongest variant is the Priority PRAM in which processors have a fixed priority according to their ID's, and the processor with the highest priority, among processors attempting to write in the same location simultaneously, succeeds.

The PRAM model is very convenient for studying the inherent parallel complexity of problems because one does not have to worry about the messy details of interprocessor communication and various distributions of data. Thus our lower bounds establish that even under ideal conditions (such as a global memory with each location accessible by all processors in unit time) there is a limit to the parallelism achievable for term matching. Our upper bounds primarily show that (in some cases) the lower bounds cannot be improved.

Before we can discuss our results, we need the following technicalities. We use superscripts (I, II, etc.) whenever we want to indicate the arities of function symbols with non-zero arity. We denote the directed edge from n_1 to n_2 by (n_1, n_2) and its label by label$((n_1, n_2))$. The label of a node n is denoted by label(n). We assume that the labeled directed tree is available in the form of an inverted tree. Such a tree is specified by three arrays: Parent, Label, and Edge_label. For each node i, Parent$[i]$ contains the node number of the parent of i, Label$[i]$ contains label(i), and Edge_label$[i]$ contains label$((i,$ Parent$[i]))$.

The assumption of inverted tree representation is for convenience only. All our lower bounds hold for uninverted trees also. In parallel computation, most algorithms for trees are more conveniently specified for inverted trees because the technique of recursive doubling or path compression is often used. Because of the fixed storage required at every vertex, inverted trees are preferred in sequential computation also. To represent uninverted trees, linked lists or arrays are required at every node, unless the awkward child–sibling binary tree representation is used. If the uninverted tree representation with linked list is used, then processor allocation alone is an onerous task, taking at least as much time as our lower bound. If arrays are used at each node, then it is easy to convert the uninverted tree into

an inverted tree, and vice versa, in constant time using polynomially many processors.

Let us assume that the two terms s and t are represented as *labeled directed trees*. Informally, the sequence of edge-labels on the path from a node to its root is referred to as its *e-path*. Term matching of s and t involves: identifying the *corresponding* nodes of the two trees representing s and t (these are nodes that have identical e-paths to their respective roots); ensuring that corresponding nodes have the same label, unless the node from the pattern is labeled by a variable, called the *homogeneity* check; and ensuring that the substitutions for different instances of the same variable are identical, called the *consistency* check.

We need the majority problem in order to obtain the lower bounds. The majority problem is defined as follows: Given n bits, $x_1, ..., x_n$, output 1 iff at least half the bits are 1's. For convenience, we shall assume that n is even throughout Section 3.

PROPOSITION 1. *The Boolean OR, AND of n^k bits stored in an array can be found in $O(1)$ time using n^k processors on the Common CRCW PRAM.*

Proof. Straightforward.

PROPOSITION 2. *The maximum of n distinct numbers stored in an array can be found in $O(1)$ time using n^2 processors on the Common CRCW PRAM.*

Proof. Omitted.

We also use the following results.

THEOREM 1 (Cook and Dwork, 1982). *The OR of n bits requires $\log_{(5+\sqrt{21})/2} n$ time on the CREW PRAM (independent of the number of processors and memory cells used).*

THEOREM 2 (Beame and Hastad, 1987). *Any CRCW PRAM requires $\Omega(\log n/\log\log n)$ time to compute the majority function of n bits, if processors are bounded by a polynomial in n.*

3.1. *Lower Bounds for the Tree and DAG Representations*

We now prove the following.

THEOREM 3. *If processors are bounded by a polynomial in n, any CRCW PRAM requires $\Omega(\log n/\log\log n)$ time for term matching, where n*

is the number of symbols in the input terms and input is in the form of two labeled ordered trees.

Proof. It suffices to show that majority can be reduced in constant time (using processors bounded by a polynomial in the input size) to term matching with trees. Therefore, let $x_3, \ldots, x_{n+2}$ be any instance of majority, where the x_i's are boolean values. We construct an instance of the term matching problem over the variables $V=\{x, y\}$, and the function symbols $F=\{f^{II}, 0^{I}, 1^{I}, a\}$, as follows (the constant a is needed to satisfy arity constraints).

First pad the input with exactly one extra 0 as x_1 and one extra 1 as x_2. Assign to each index $i \in I_{n+2}$, $(i-1)^2$ processors. These processors find in constant time the maximum index j, $1 \leqslant j < i$ such that $x_j = x_i$ (Proposition 2). Once such an index is found, for each index i in $I_{n+2} - I_2$, a representative processor p_i stores j in Parent$[i]$. Parent$[0]$ = Parent$[1]$ = Parent$[2]$ = 0. Parent$[n+3] = \max\{j \mid x_j = 1\}$ and Parent$[n+4] = \max\{j \mid x_j = 0\}$. Now for each $i \in I_{n+2}$, Label$[i]$ is set to x_i. Label$[n+3]$ = Label$[n+4] = a$. A third array called Edge_label$[1 : n+4]$ is set to one for each i such that Parent$[i] \neq 0$. Edge_label$[1]$ is set to 1 and Edge_label$[2]$ is set to 2. Label$[0]$ is set to f, i.e., f is the label of the root.

The three arrays constitute the subject in the form of a labeled inverted ordered tree. The pattern is always a fixed (i.e., depends only on n) term represented by a tree with the root vertex labeled f, the first child (edge_label 1) of the root is a vertex labeled x, and the second child is a chain of $n/2+2$ vertices, each labeled 1, except the last which is labeled y. The label of each edge in the chain is 1 (see Fig. 1). We note that both terms can be constructed by $O(n^3)$ processors in $O(1)$ time, on the Common CRCW PRAM. It is now easy to see that the two terms, so constructed, match iff the output for the corresponding instance of majority is 1. The lower bound now follows from Theorem 2. ∎

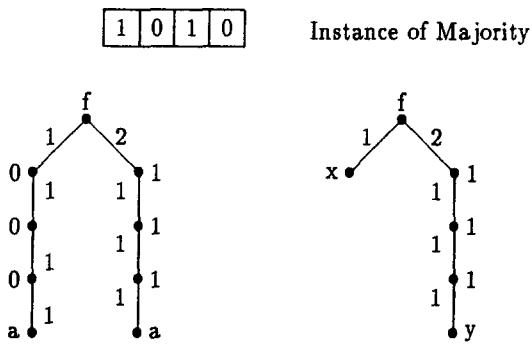

FIG. 1. The two labeled ordered trees corresponding to the instance of Majority.

Several interesting features of this proof deserve further mention. Clearly, term matching with linear terms also has the same lower bound. If we allow two forests as input, then term matching with *just one variable* is at least as hard as majority. Note that the proof does not in any way exploit the time that must be spent for the consistency check. In fact, the lower bound holds for any parallel algorithm that takes two ordered trees and finds all pairs of corresponding nodes. Since the reduction can be carried out on the weakest CRCW PRAM, the theorem holds for all CRCW PRAM's.

THEOREM 4. *The lower bound, on time, of Theorem* 3 *holds for labeled tree isomorphism also.*

Proof (Sketch). The subject is constructed as in Theorem 3 above. We construct $n/2+1$ patterns, $P_{n/2+1}, \ldots, P_{n+1}$, where pattern P_i consists of a root vertex labeled f and two subtrees. The left subtree of the root is identical to the left subtree of the subject, and the right subtree is a chain of $i+1$ vertices, each labeled 1, except the last which is labeled by the constant a. The entire construction can be done in constant time using polynomially many processors. By Proposition 1, we can OR the results of the isomorphism tests for these $n/2+1$ patterns against the subject, in constant time. It is easy to see that the output of the OR operation is a 1 iff at least half the bits are 1's in the corresponding instance of majority. ∎

The lower bound for term matching must be contrasted with the constant time upper bound for string matching on the CRCW PRAM, using as many processors as the product of the lengths of the two strings. Hence, in general, problems with the tree representation may require more parallel time compared to those using the string representation. We have given one specific example where this is actually the case. One of the reasons for this difference is that the tree representation of a term is not unique. One could therefore ask whether the lower bound for matching still holds, if each term has a unique representation. The answer is yes, as we show in the next section that the lower bound for term matching holds for the string representation, which is unique. As a consequence of Theorems 3 and 4, we have

COROLLARY 1. *The lower bound on parallel time for majority also holds for* (1) *list ranking*, (2) *finding the height of an inverted tree, and* (3) *computing the level of a node in an inverted tree.*

Proof. This result follows from an examination of the proofs of the previous theorems. For a definition of list ranking, see Cole and Vishkin, (1986). ∎

For the EREW and CREW models we have the following theorem.

THEOREM 5. *Term matching with trees on the EREW or CREW PRAM requires* $\Omega(\log n)$ *time.*

Proof. We give a constant time reduction from the (boolean) AND problem to term matching with trees. The subject consists of a root r labeled f_n with n children, $v_1, v_2, ..., v_n$, corresponding to the n input values $x_1, x_2, ..., x_n$. For each i, vertex v_i is labeled x_i, and the label of the edge from v_i to r is i. The pattern consists of a root labeled f_n with n children, each labeled 1, and again the edge labels range from 1 to n. It is easy to see that the two terms can be constructed by n processors in constant time on the EREW PRAM. The two terms, so constructed, match iff the output is 1 for the corresponding instance of the AND problem. The lower bound now follows from Theorem 1. Note that by a simple fanout tree construction, we can do without an infinite set of function symbols f_n, one for each n. Specifically, the construction can be modified so as to require only one function symbol f of arity 2 besides the constants 0 and 1. ▮

COROLLARY 2. *The lower bound of Theorem* 5 *also applies to* (1) *the equivalence problem for two ground terms,* (2) *pattern matching on strings, and* (3) *labeled ordered tree isomorphism.*

Proof. Observe that both trees constructed in the proof of Theorem 5 are ground, i.e., they contain no variables. ▮

We conclude this section by observing that our lower bounds also hold for the DAG representation.

THEOREM 6. *The lower bounds for term matching also hold for the DAG representation.*

Proof. Observe that the DAG representations of the trees that we have constructed in the above proofs, corresponding to the given instance of majority, cannot be more compact (except by a constant factor). Thus any algorithm for term matching with DAGs must spend at least as much time as our lower bound for trees. ▮

3.2. *Lower Bounds for the String Representation*

When terms are represented as strings stored in arrays the lower bound is relatively more difficult to obtain. The difficulty lies in computing, in constant time, the location where a symbol must be stored in the array. It turns out that this is not very difficult, if we allow a "reasonably" small

number of spaces in the strings representing the two terms or at most n redundant trailing right parentheses. However, we shall not make such assumptions. Although, the lower bound for term matching with strings can be obtained by using the weaker notion of constant-depth truth-table reducibility (as in the proof for labeled tree isomorphism) we shall prove a stronger result, viz., majority can be (many-one) reduced in constant time to term matching with strings. In the following theorem, the model is a Common CRCW PRAM.

THEOREM 7. *If processors are bounded by a polynomial in n, any CRCW PRAM requires $\Omega(\log n/\log\log n)$ time for term matching, where n is the number of symbols in the input terms and the input is in the form of two strings stored in arrays.*

Proof. Let $x_1, ..., x_n$ be an instance of majority. Without loss of generality assume that $x_1 = 1$ and $x_n = 0$ (we can always pad the input with a 0 and a 1 so that this is true). We construct the subject t, stored in array $T[6 : 13n+2]$, and pattern s, stored in array $S[1 : 7n/2 - 1]$, as follows. The set of variables is $V = \{y_1, ..., y_{n/2+1}\}$ and the set of function symbols is $F = \{h^{\text{II}}, f^{\text{I}}, k^{\text{I}}, a\}$. We also use an array Temp$[1 : 2n]$. All arrays are initialized to the character # in $O(1)$ time using $O(n)$ processors.

Now for $i \in I_n$, processor p_i reads x_i and stores it in Temp$[n(1-x_i)+i]$. Thus we have separated the 1's from the 0's. For $i \in I_{2n}$, processor p_i reads Temp$[i]$ and does the following. If Temp$[i] \neq \#$, $i < 2n$, and $x_i = 1$, then store h in $T[6i]$, a left parenthesis in $T[6i+1]$, and f in $T[6i+2]$. If Temp$[i] \neq \#$, $i < 2n$, and $x_i = 0$, then store h in $T[6i]$, a left parenthesis in $T[6i+1]$, and k in $T[6i+2]$. Processor p_{2n} stores the term $k(a)$ in locations $T[12n]$ to $T[12n+3]$. Locations $T[12n+4]$ through $T[13n+2]$ are set to right parentheses (note there are exactly $n-1$ occurrences of h).

Now, using $O(n^3)$ processors overall, the distance between each occurrence of the symbols f or k and the nearest h (if any) is found (from Proposition 2, this can be done in constant time). Note that all these distances are multiples of 3. With $O(n^2)$ processors, in constant time, all these distances are padded with appropriate-sized strings of the form $(f(... f(a) ...))$ (e.g., for a distance of 3 use (a), for 6 use $(f(a))$, etc.). This completes the construction of the subject, which has a many occurrences of f (as children of h) as there are 1's in the instance of majority and as many k's as 0's. The pattern is always a fixed term of the form $h(f(y_1), ..., h(f(y_{n/2}), y_{n/2+1}) ...)$, with $n/2$ occurrences of f. It is easy to see that it can also be constructed in constant time with $O(n)$ processors. Now we claim that s matches t iff at least half the bits are 1's in the corresponding instance of majority. ∎

The $\Omega(\log n/\text{loglog } n)$ time lower bound for term matching with strings, on all CRCW PRAMs, follows from this theorem. It is easy to see that the reduction on the EREW model, given in the previous section, can be modified to output terms in the form of strings stored in arrays. Hence the $\Omega(\log n)$ time lower bounds for term matching with strings follow on the CREW and EREW models.

3.3. *Optimal Time Upper Bounds*

In this section we show how to improve the upper bounds of Verma *et al.* (1986) and Ramesh *et al.*(1989) on the CREW model, when terms are represented as trees. We also design optimal time algorithms for the Arbitrary (and hence Priority also) CRCW PRAM and the CREW PRAM, when terms are represented as strings stored in arrays. To the best of our knowledge these are the first optimal algorithms for term matching with the string representation. We shall first present optimal algorithms for the string representation. To derive the optimal algorithm (on the CREW model) for the tree representation, we get rid of the relatively complicated processor allocation problem by linearizing the two trees into strings and then invoking our optimal algorithm for strings. We note that a direct algorithm is also possible. Our algorithms use the sublogarithmic time algorithms in Cole and Vishkin (1986) for computing prefix sums is parallel. In Cole and Vishkin (1986) it is shown that the prefix sums of n numbers, each of $\log n$ bits, can be computed in time $O(\log n/\text{loglog } n)$ using n loglog $n/\log n$ processors.

Just as there is a notion of corresponding nodes in the tree representation, there is an analogous notion of corresponding indices in the string representation. The difference lies in the way we identify the corresponding indices. The input terms to the algorithm are stored in two arrays, S containing the subject and P containing the pattern.

1. {For each left (right) parenthesis find its nesting level and the index of the corresponding right (left) parenthesis.} Assign to each symbol a value 0, $+1$, or -1 in array B as described in Table 1. A, in Table 1, is just a temporary array into which S has been copied. Now the parallel prefix algorithm of Cole and Vishkin (1986) is used to compute the nesting level of each symbol in S. Sort all parentheses on the pair (nesting level, index in S). The corresponding parentheses are now in adjacent locations. For each parenthesis store the location of its corresponding parenthesis in an array.

2. {For each index of S containing a non-parenthesis symbol find its order with respect to the nearest left parenthesis (if any) enclosing it.} For example, if $f(a, g(b, b), a)$ is stored without commas in $S[1 : 10]$, the order

TABLE 1

Table Used in Step 1

Symbol in $A[i-1]$	Symbol in $A[i]$	$B[i] \cdot level$
★	★	0
★	(	+1
★	)	0
(	★	0
(	(	not possible
(	)	not possible
)	★	−1
)	)	−1
)	(	not possible

Note. ★ refers to any non-parenthesis entry.

of indices 4 and 7 is 2. For each pair of corresponding parentheses, copy all symbols immediately nested in this pair into a separate array. Each symbol is assigned the value 1 and the parallel prefix algorithm is used to determine the order of each index within the nearest enclosing parentheses pair.

3. {Obtain the e-paths, to the root, of all indices of S containing non-parenthesis symbols.} For each index i containing a non-parenthesis symbol s_i, find all the left parentheses in which it is nested and get the order of the indices which are immediate predecessors of these parentheses. Observe that s_i is nested inside left parenthesis l_j, if $j < i <$ index (corresponding right parenthesis of l_j). Store this information in a separate array for each i. This will be referred to as the e-path of index i, written e-path(i).

4. Repeat the above three steps for the pattern.

5. {Find corresponding indices.} For each index i of P containing a non-parenthesis symbol p_i, find the index j of S such that e-path(i) = e-path(j). To each index in the pattern there corresponds exactly one index in the subject with the same e-path. If for some index in the pattern there is no such index in the subject, then halt and output "No match."

6. Once the corresponding indices have been determined, the homogeneity check is trivial.

7. {Consistency check.} Determine all indices of the same variable in the pattern. Since the corresponding indices in the subject have been determined, we just have to ensure that the terms beginning at these indices in the subject are syntactically identical.

We note that some of the ideas here have been used in Ramesh *et al.* (1989) to convert terms represented as strings into trees. However, their

ideas do not yield tight upper bounds. The proof of correctness is straightforward and therefore omitted.

Complexity on the Arbitrary CRCW PRAM. There is an obvious implementation of this algorithm which requires $O(n^3)$ processors and $O(\log n/\log\log n)$ time, but we show that $O(n^2 \log n)$ processors suffice to meet the stated time bound. For Step 1, we do not explicitly sort the pairs. Instead, in an array Temp$[1:n]$, processor p_i stores the level of symbol $S[i]$ in Temp$[i]$, if $S[i]$ is a parenthesis (Temp$[i]$ is 0 otherwise). Now for each j such that Temp$[j] \neq 0$ and $S[j]$ is a left parenthesis, we can find the minimum element of the set $\{i \mid \text{Temp}[i] = \text{Temp}[j] \wedge S[i] = ')' \wedge i > j\}$ with $O(n)$ processors in $O(\log\log n)$ time, using the algorithm of Shiloach and Vishkin (1981). Thus Step 1 requires $O(n^2)$ processors and $O(\log n/\log\log n)$ time. It is not very difficult to check that Steps 2, 3, 6, and 7 can also be done within these bounds. Step 5 is a bit trickier. We use the location in the array and the level of each symbol to sort the e-paths. This requires another application of parallel prefix. Now assign to each symbol P_i in the pattern $\log n \times \text{level}(P_i)$ processors. Divide the symbols at level(P_i) in the subject into $\log n$ groups of size $n/\log n$. Each set of level(P_i) processors compares e-path(P_i) against the e-path of one symbol in each group in constant time. The $\log n$ results are collected in constant time and the search (if unsuccessful) narrows down to one of the groups of size $n/\log n$. Thus in time $O(\log n/\log\log n)$ all the corresponding symbols have been identified. Thus Step 5 requires at most $n^2 \log n$ processors and we are done.

Complexity on the CREW PRAM. We note that there is an algorithm for parallel prefix on the CREW PRAM which requires n processors and $O(\log n)$ time (Kruskal *et al.*, 1985). Thus we have the following result: There is an algorithm for term matching with strings which requires $O(n^3)$ processors and runs in $O(\log n)$ time. In fact, for any positive constant ε, $0 < \varepsilon < 1$, there is an algorithm that runs in $O((\log n)/\varepsilon)$ time and requires $n^{2+\varepsilon}$ processors. This can be done by assigning to each symbol in the pattern $n^{1+\varepsilon}$ processors for Step 5, which dominates the complexity.

TABLE 2

Table Summarizing Lower Bounds on Parallel Term Matching

D.S.	Model	Lower Bound (time, processors)
Trees	Any CRCW	$\Omega(\log n/\log\log n)$, poly
Strings	Any CRCW	$\Omega(\log n/\log\log n)$, poly
Trees	C(E)REW	$\Omega(\log n)$, unbounded
Strings	C(E)REW	$\Omega(\log n)$, unbounded

TABLE 3

Table Summarizing Upper Bounds on Parallel Term Matching

D.S.	Model	Upper Bound (time, processors)
Trees	ARBITRARY	$O(\log n)$, n
Strings	ARBITRARY	$O(\log n/\log\log n)$, $n^2 \log n$
Trees	C(E)REW	$O((\log n)/\varepsilon)$, $n^{2+\varepsilon}$
Strings	C(E)REW	$O((\log n)/\varepsilon)$, $n^{2+\varepsilon}$

For the tree representation, we use the algorithm devised in Ramesh and Ramakrishnan (1987) for converting trees into strings. On the CREW model this conversion can be done by n processors in $O(\log n)$ time. Thus we have an optimal time algorithm for term matching with trees, on the CREW model also. We note that a direct optimal time algorithm is also possible, but we do not discuss it here.[2]

These complexity results are summarized in Tables 2 and 3.

4. Associative Commutative Matching

So far we have assumed that each function symbol has a fixed arity. This assumption is inconvenient for terms containing associative function symbols. A term that has some associative functions symbols is more conveniently represented in "flattened" form. For example, $f(a, f(b, c))$ is flattened to $f(a, b, c)$, if f is an associative function symbol (i.e., f can have any arity greater than 1). Although this notion is very intuitive, our proofs require a formal definition. The following deals with the required formalism.

A *string* is any finite sequence of symbols from $V \cup F \cup \{(\ ,\ )\}$. We are interested in certain kinds of strings: the expressions. For each n-ary (binary) non-associative (associative) function symbol $f \in F$, we define an n-ary (of arity $\geqslant 2$) expression-building operation $\mathscr{F}_f$ on strings: $\mathscr{F}_f(u_1, ..., u_n) = f(u_1, ..., u_n)$. To be precise we should have written $\mathscr{F}_f((u_1, ..., u_n))$ but, as is customary, we shall omit the extra parentheses.

Definition 4 (Expressions). *The set $\mathscr{E}$ of expressions is the set of strings generated from the constants and variables by the $\mathscr{F}_f$ operations.*

Our definition of expressions is motivated by the need to introduce flattening of terms when function symbols are associative. Informally,

[2] This algorithm requires n^2 processors and $O(\log n)$ time.

to flatten a term with respect to a function symbol f represent it in right associative form. The term will now be of the form $f(t_1, f(t_2, ..., f(t_{n-1}, t_n)...)$, where the t_i's do not start with f. Now represent the term as $f(t_1, ..., t_n)$.

Notation. The flattened form of any expression ε will be denoted by $\bar{\varepsilon}$.

We require the following auxiliary definitions for flattening. For $n > 0$, let $\mathscr{E}^n = \{(\varepsilon_1, ..., \varepsilon_n) \mid \varepsilon_i \in \mathscr{E} \text{ for } i \in I_n\}$, $\mathscr{E}^0 = \{(\,)\}$ and $\mathscr{E}^+ = \bigcup_{i=1}^{\infty} \mathscr{E}^i$. Let $\bigstar$ be a new symbol not in $V \cup F$.

DEFINITION 5. $\arg: \mathscr{E} \to \mathscr{E}^+$ and $\mathrm{op}: \mathscr{E} \to V \cup F \cup \{\bigstar\}$ are defined as follows:

1. $\arg(v) = (v)$, $\mathrm{op}(v) = \bigstar$, $\arg(a) = (a)$ and $\mathrm{op}(a) = a$, v any variable, a any constant.

2. if $\varepsilon = f(\varepsilon_1, ..., \varepsilon_n)$, $n \geq 1$, $f \in F$, then $\arg(\varepsilon) = (\varepsilon_1 \cdots \varepsilon_n)$ and $\mathrm{op}(\varepsilon) = f$.

Define $\mathrm{rem}: \mathscr{E} \times F \to \mathscr{E}^+$ as $\mathrm{rem}(\varepsilon, p) = \arg(\varepsilon)$ if $\mathrm{op}(\varepsilon) = p$ and p is associative, (ε) otherwise. Finally, for each $n \geq 2$, let con_n be the operation which takes n tuples from $\mathscr{E}^+$ and returns the tuple formed by "concatenating" the n tuples in order. Since expressions are defined inductively, our definition of flattening is recursive.

DEFINITION 6 (Flattening). 1. For each variable v, $\bar{v} = v$.

2. if $\varepsilon = f(\varepsilon_1, ..., \varepsilon_n)$ $(n \geq 2)$ and f is associative then $\bar{\varepsilon} = \mathscr{F}_f(\mathrm{con}_n(\mathrm{rem}(\overline{\varepsilon_1}, f), ..., \mathrm{rem}(\overline{\varepsilon_n}, f)))$.

3. if $\varepsilon = f(\varepsilon_1, ..., \varepsilon_n)$ $(n \geq 0)$ and f is non-associative then $\bar{\varepsilon} = f(\overline{\varepsilon_1}, ..., \overline{\varepsilon_n})$.

We illustrate flattening of $f(a, f(b, c))$ based on the above definition:

$$\begin{aligned}
\overline{f(a, f(b, c))} &= \mathscr{F}_f(\mathrm{con}_2(\mathrm{rem}(\bar{a}, f), \mathrm{rem}(\overline{f(b, c)}, f))) \\
&= \mathscr{F}_f(\mathrm{con}_2((a), \mathrm{rem}(\mathscr{F}_f(\mathrm{con}_2(\mathrm{rem}(\bar{b}, f), \mathrm{rem}(\bar{c}, f)))))) \\
&= \mathscr{F}_f(\mathrm{con}_2((a), \mathrm{rem}(f(b, c), f))) \\
&= \mathscr{F}_f(\mathrm{con}_2((a), (b, c))) \\
&= f(a, b, c).
\end{aligned}$$

The set of flattened terms $\bar{\mathscr{T}}$ is exactly the set of expressions obtained by flattening terms. We define a flattened substitution (such a substitution will be denoted with a bar on top) analogous to the definition of a substitution, except that its range is $\bar{\mathscr{T}}$ instead of $\mathscr{T}$. The following lemmas are easy consequences of our definition of flattening.

LEMMA 1. *For any term $t \notin V$ and any substitution σ,*

1. $|\arg(\bar{t})| = |\arg(t)| = |\arg(\overline{\sigma(t)})|$ *if* $\mathrm{op}(t)$ *is non-associative.*
2. $|\arg(t)| \leqslant |\arg(\bar{t})| \leqslant |\arg(\overline{\sigma(t)})|$ *otherwise.*

LEMMA 2. *For any boolean term $t \notin V$ and any substitution σ,*

1. $|\arg(\bar{t})| = |\arg(t)| = |\arg(\overline{\sigma(t)})|$ *if* $\mathrm{op}(t)$ *is non-associative.*
2. $|\arg(\bar{t})| = |\arg(\overline{\sigma(t)})|$ *otherwise.*

An interesting property of boolean terms is the following:

LEMMA 3 (Distribute). $\overline{\sigma(t)} = \bar{\sigma}(\bar{t})$ *for any substitution σ and boolean term t.*

Remark 2. In a tree or DAG representation of a term, the children of a node labeled by an associative function symbol are ordered. This, however, is not the case when the function is commutative. In the rest of this paper, we shall assume that all terms containing associative terms have been flattened. This assumption is just for convenience. The process of flattening a term containing associative operators takes only linear time.

We use the following results in our proofs. The first result relates the space used by a non-deterministic Turing machine accepting a set A to the depth needed of a uniform circuit accepting A. The second and third relate the complexity of rooted subtree isomorphism to that of bipartite matching. In *rooted subtree isomorphism*, we are given two rooted trees $T_1 = (V_1, E_1, r_1)$ and $T_2 = (V_2, E_2, r_2)$ and we want to know whether there is an isomorphism $\psi : V_1 \to V$, $V \subseteq V_2$, such that (i) $\psi(r_1) = r_2$ and (ii) $(v_i, v_j) \in E_1$ iff $(\psi(v_i), \psi(v_j)) \in E_2$.

THEOREM 8 (Borodin, 1977). *Let A be a language recognized by a non-deterministic $S(n)$ space bounded Turing machine where $S(n) \geqslant \log n$. Then there exists $d > 0$ such that $DEPTH_A(n) \leqslant dS(n)^2$.*

THEOREM 9 (Reyner, 1977). *Let $G = (V_1, V_2, E)$ be a bipartite graph with $r = |V_1|$, $s = |V_2|$, and $r \leqslant s$. Let p and q be the two trees for subtree isomorphism with sizes n and m (p is to be mapped into q). Given an algorithm for bipartite matching which requires at most $O(rs)$ operators, the subtree isomorphism algorithm requires at most $O(nm \log n)$ operations.*

THEOREM 10 (Verma, 1988; Verma and Reyner, 1989). *Let $G = (V_1, V_2, E)$ be a bipartite graph with $r = |V_1|$, $s = |V_2|$, and $r \leqslant s$. Let p and*

q be the two trees for subtree isomorphism with sizes n and m (p is to be mapped into q). Given an algorithm for bipartite matching which requires at most $O(rs^u)$ operations for $u > 1$, the subtree isomorphism algorithm requires at most $O(nm^u)$ operations.

4.1. *Sequential Complexity*

4.1.1. *2-Occurrence Associative Commutative Matching*

We show that 2-occurrence AM, 2-occurrence CM, and 2-occurrence ACM are NP-complete (and hence so is 2-occurrence GACM). This characterizes completely the sequential complexity of AM, CM, and ACM when variable occurrences are varied. Membership in NP of all these problems follows from that of the general problem (Benanav *et al.*, 1985), so we omit this part in all our proofs of NP-completeness.

THEOREM 11. *2-occurrence AM is NP-complete.*

Proof. Let $C = \{c_1, ..., c_m\}$ be an instance of SAT over the boolean variables $x_1, ..., x_n$, $m, n > 0$, such that $|c_j| \leqslant 3$ and each x_i is restricted to at most 3 occurrences (in negated or unnegated form). This problem is known to be NP-complete (Garey and Johnson, 1979). Without loss of generality we can assume that $2 \leqslant |c_j| \leqslant 3$. For let c_j be any clause of size 1, so $c_j = z_i$ where z_i is some literal. We set z_i to the appropriate value to satisfy c_j and obtain a simpler instance C', in which neither c_j nor z_i appears, such that C' is satisfiable iff C is satisfiable. Now, given C we construct two flattened terms s and t over $F = \{g, f, h, 0, 1\}$, where f is of arity 6, g is associative, h is of arity m, and 0 and 1 are constants. Let $V = \{x_1, ..., x_n\} \cup \{y_1, ..., y_n\} \cup \{u_{ij} \mid i \in I_m, j \in I_6\}$. The main ideas of the reduction are as follows. The truth and falsity of a boolean variable are simulated by the substitutions $x_i = 1$ and $x_i = 0$, respectively. Each variable y_i simulates the role of $\overline{x_i}$ and the u_{ij}'s are dummy variables. Without loss of generality, we can assume that every variable in C occurs at most once negated, at most twice unnegated, and at least once unnegated. On the first occurrence of a variable x_i in unnegated form (the first occurrence of x_i is given by the least j such that $c_j \in C$ contains x_i), we use one occurrence of y_i to ensure that y_i is instantiated to $\overline{x_i}$. The second occurrence of y_i is used whenever $\overline{x_i}$ occurs in the instance of SAT. For each clause c_j such that $|c_j| = 3$ we do the following: let x_1, x_2, and x_3 be the variables in c_j. Let $q_1, ..., q_7$ be 7 distinct terms, which represent the 7 truth assignments that satisfy clause c_j, obtained by defining $q_i = f(b_1, 1 - b_1, b_2, 1 - b_2, b_3, 1 - b_3)$, for $i \in I_7$. Here, for $k \in I_3$, $b_k \in \{0, 1\}$ and the assignment (b_1, b_2, b_3) satisfies c_j. Let $t_j = g(a, q_1, ..., q_7, a)$ and $s_j = g(u_{j1}, f(l_1, m_1, l_2, m_2, l_3, m_3), u_{j2})$, where for $i \in I_3$, $l_i = x_i$ if x_i occurs unnegated in clause

c_j, otherwise $l_i = y_i$; $m_i = y_i$ if c_j is the first clause in which x_i appears unnegated, otherwise $m_i = u_{j(i+2)}$. Similarly, for each clause c_j such that $|c_j| = 2$, let x_1, x_2 be the variables in c_j. Let q_1, q_2, q_3 be 3 distinct terms, which represent the 3 truth assignments that satisfy c_j, obtained by defining q_i as above. Let $t_j = g(a, q_1, q_2, q_3, a)$ and $s_j = g(u_{j1}, f(l_1, m_1, l_2, m_2, u_{j5}, u_{j6}), u_{j2})$, where the l_i's and m_i's, for $i \in I_2$, are defined as above. Finally let $s = h(s_1, ..., s_m)$ and $t = h(t_1, ..., t_m)$ (see Fig. 2). It is not difficult to see that s a-matches t iff C is satisfiable, and that each variable $v \in V$ appears at most twice. ∎

THEOREM 12. *2-occurrence CM is NP-complete.*

Proof (Sketch). The proof is very similar to the one given above, except for the structure of the terms s_j and t_j. Of course, in this case g is commutative (and not associative as in the previous proof). The same restricted version of SAT is used here. For c_j such that $|c_j| = 3$, let $t_j = g(g(g(q_1, q_2), g(q_3, q_4)), g(g(q_5, q_6), g(q_7, a)))$, where q_i's are defined as in Theorem 11 and

$$s_j = g(u_{j1}, g(u_{j2}, g(u_{j3}, f(l_1, m_1, l_2, m_2, l_3, m_3)))).$$

As before, $l_i = x_i$ if x_i occurs unnegated in clause c_j, otherwise $l_i = y_i$; m_i is y_i if c_j is the first clause in which x_i appears unnegated, otherwise $u_{j(i+3)}$. Note that the set of dummy variables is larger here. The construction is similar for clauses of length two. Finally, let $s = h(s_1, ..., s_m)$ and $t = h(t_1, ..., t_m)$. ∎

COROLLARY 3. *2-occurrence ACM and 2-occurrence GACM are NP-complete.*

Proof. Observe that 2-occurrence AM is just a special case of 2-occurrence GACM. The NP-hardness of 2-occurrence ACM is obtained from the reduction of Theorem 11, by assuming that g is AC. ∎

$$C = \{c_1 = x_1 \vee x_2 \vee x_3,\ c_2 = \overline{x_1} \vee \overline{x_2} \vee x_3,\ c_3 = x_1 \vee x_2 \vee \overline{x_3}\}$$

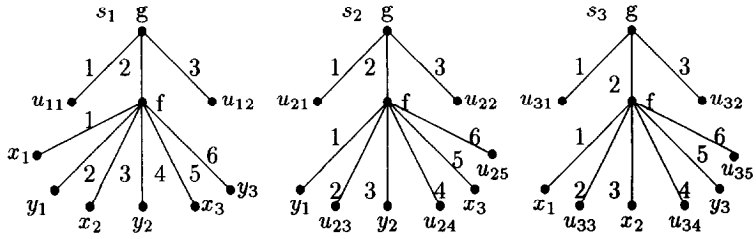

FIG. 2. The three subtrees of the pattern, $s = h(s_1, s_2, s_3)$, corresponding to C.

4.1.2. *Boolean Associative Commutative Matching*

We show that even under the simplification of boolean terms, CM is NP-complete whereas AM has a linear-time algorithm. Boolean terms are defined as follows.

DEFINITION 7.

- A variable or a constant is a term, and
- if $f \in F$ is neither associative nor commutative and $t_1, ..., t_{a_f}$ are terms then so is $f(t_1, ..., t_{a_f})$;
- if $f \in F$ is either associative or commutative and t_1, t_2 are terms not belonging to the set of variables, then so is $f(t_1, t_2)$.

Boolean CM now refers to the problem of commutative matching when the set of operators in the boolean terms consists of some free and some commutative operators. Boolean AM is defined similarly. As before, boolean GACM is the most general problem, where the set of boolean terms can conatin all kinds of operators: free, associative only, commutative only, or both associative and commutative. It is easy to see that a flattened boolean term retains the property that no interpreted operator can have a variable as its argument. As before, we assume that all boolean terms are given in flattened form.

THEOREM 13. *Boolean CM is NP-complete.*

Proof (Sketch). NP-hardness of Boolean CM is proved as follows. Let $C = \{c_1, ..., c_m\}$ be an instance of 3SAT over the n boolean variables, $x_1, ..., x_n$, $m, n > 0$. Let $F = \{f, g, a, h, p, 0, 1\}$, where f is a binary commutative function symbol, g is a ternary function symbol, p is a unary function symbol, a, 0, and 1 are constants, and h is an m-ary function symbol. Let $V = \{x_1, ..., x_n\} \cup \{u_{ij} \mid i \in I_m, j \in I_7\}$. The truth and falsity of a boolean variable x_i is simulated by the substitution of 1 and 0 respectively for x_i. For each clause c_j, do the following. Let x_1, x_2, and x_3 be the variables in c_j. Let $q_1, ..., q_7$ be 7 distinct terms, which represent the 7 truth assignments that satisfy clause c_j, obtained by defining $q_i = g(b_1, b_2, b_3)$, where $b_i \in \{0, 1\}$ and the assignment (b_1, b_2, b_3) satisfies c_j. Let

$$s_j = f(f(f(p(g(x_1, x_2, x_3)), p(u_{j1})), f(p(u_{j2}), p(u_{j3}))),$$
$$f(f(p(u_{j4}), p(u_{j5})), f(p(u_{j6}), p(u_{j7}))))$$

and

$$t_j = f(f(f(p(q_1), p(q_2)), f(p(q_3), p(q_4))),$$
$$f(f(p(q_5), p(q_6)), f(p(q_7), p(a)))).$$

Finally, let $s = h(s_1, ..., s_m)$ and $t = h(t_1, ..., t_m)$. (See Fig. 3.) It is easy to see that s can be commutative-matched with t iff C is satisfiable. ▮

COROLLARY 4. *Boolean 2-occurrence CM, boolean 2-occurrence ACM, and boolean 2-occurrence GACM are NP-complete.*

Proof. The reduction of Theorem 13 can be modified to use the restricted version of SAT given in Theorem 11, instead of 3SAT, by employing the technique of Theorem 12 to save variable occurrences. ▮

However, associative matching of boolean terms requires only time linear in the size of the input.

THEOREM 14. *If s and t are any two boolean terms and σ_1 and σ_2 any two substitutions such that $\sigma_1(s) =_A \sigma_2(s) =_A t$, then $\overline{\sigma_1}|_{V_s} = \overline{\sigma_2}|_{V_s}$.*

Proof (Sketch). Fix an s. Our proof is by induction on the height of t.

Basis. t is any constant. If $s \in V$ then $\sigma_1(s) = \sigma_2(s) = t \Rightarrow \overline{\sigma_1}|_{V_s} = \overline{\sigma_1}|_{\{s\}} = \overline{\sigma_2}|_{V_s} = t$. If s is a constant, then $V_s = \varnothing$. Therefore, $\overline{\sigma_1}|_{V_s} = \overline{\sigma_2}|_{V_s}$ trivially, for any σ_1 and σ_2. If $s = f(...)$, there is no substitution which matches s to t.

Induction step. $t = f(t_1, ..., t_n)$ and there are two cases to be considered:

Case 1. f is non-associative. If s is a variable or a constant there is nothing to prove. If $s = g(s_1, ..., s_m)$ and $f \neq g$, then also we are done. Finally, if $s = f(s_1, ..., s_m)$, then since f is non-associative we must have $m = n$. Now, if σ_1 and σ_2 are such that $\sigma_1(s) = \sigma_2(s) = t$, then

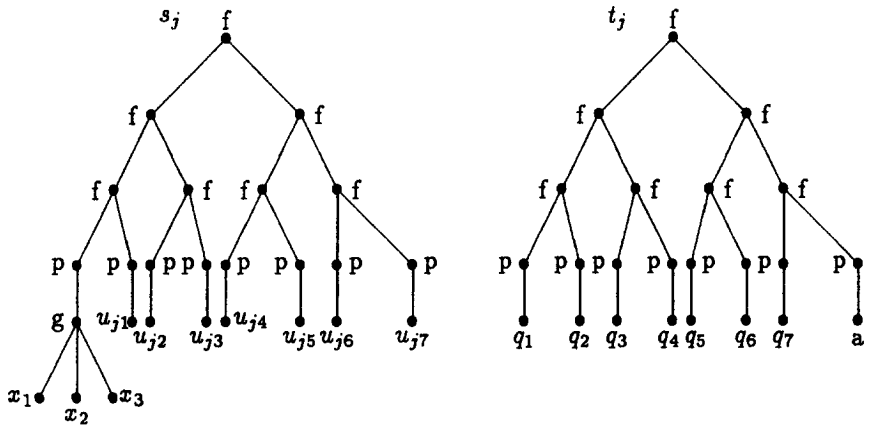

FIG. 3. Illustration for Theorem 13.

$f(\sigma_1(s_1), ..., \sigma_1(s_n)) = f(\sigma_2(s_1), ..., \sigma_2(s_n)) = f(t_1, ..., t_n)$, which implies that $\sigma_1(s_i) = \sigma_2(s_i) = t_i$ for all $i \in I_n$ (f is non-associative). From the induction hypothesis it follows that $\overline{\sigma_1}|_{V_{s_i}} = \overline{\sigma_2}|_{V_{s_i}}$ which implies $\overline{\sigma_1}|_{V_s} = \overline{\sigma_2}|_{V_s}$ (σ_1 and σ_2 are substitutions and $V_s = \bigcup_{i=1}^{n} V_{s_i}$).

Case 2. f is associative. As before, $s = f(s_1, ..., s_m)$ otherwise there is nothing to prove. Now for any σ, $|\arg(\overline{\sigma(s)})| = |\arg(\bar{s})| = |\arg(s)| = m$ ($s \in \mathscr{T}$) by Lemma 2, whence we must have $m = n$ or we are done. Now by Lemma 3, and an argument similar to the one in case 1, we establish the induction for case 2.

Since s was arbitrary the theorem is proved. ▮

From the proof of this theorem it is clear that the linear time algorithm for term matching can be used for boolean AM also. In fact, Theorem 14 is a special case of a much stronger theorem that we state without proof.

THEOREM 15. *For any boolean terms $s, t \in \mathscr{T}$ containing associative or uninterpreted operators only, there is at most one most general unifier σ (up to variable renaming).*

From these two theorems and the fact that term matching is in AC^1 (Ramesh *et al.*, 1989) we have the following results:

COROLLARY 5. *Boolean AM and Boolean AU have linear time algorithms.*

Proof. Use the linear time algorithm of Paterson and Wegman (1978) for uninterpreted unification. ▮

COROLLARY 6. *Boolean AM is in $AC^1 \subseteq NC^2$.*

4.1.3. *Linear Associative Commutative Matching*

In Benanav *et al.* (1985) it was shown that ACML $\in$ P through a natural recursive reduction of ACML to bipartite matching. Unfortunately, the depth of this reduction is proportional to the height of the input trees, and hence it is not an NC reduction. It was also claimed in Benanav *et al.* (1985) that the time complexity of this reduction is $O(|s|\ |t|^3)$, where s is the pattern and t the subject. We present a slightly different algorithm for GACML and show that it is more efficient, using the very general result of the first author (Theorem 10). Observe that because of the linearity requirement there is no need for a consistency check. Also note that for linear terms, AC matching is very similar to rooted subtree isomorphism. We have the same freedom in mapping vertices to vertices in subtree isomorphism that is allowed by an AC operator for mapping its arguments. This is explained below.

Let S and T be the two given trees, and let the problem be to determine whether S is isomorphic to a subtree of T at the root. The basic idea is that subtree isomorphism is trivial when the number of vertices in $S=(V, E)$ is at most two. If $|V| \leqslant 2$, then S is isomorphic to any rooted tree T with at least as many vertices as S. Therefore, we determine for each vertex in the two trees, S and T, the number of its descendants. It should be clear that this takes only $O(n+m)$ time, where $n=|S|$ and $m=|T|$, $n \leqslant m$. Let $D(v)$ denote the number of descendants of vertex v (including itself). A high-level description of the algorithm is as follows. It is assumed that the algorithm is invoked with (sub)trees having at least 3 vertices. If G is any graph, let $V(G)$ denote the number of vertices of G.

ALGORITHM. Rooted Subtree Isomorphism. (1) If $V(S) \leqslant V(T)$ then delete the roots of S and T to obtain rooted subtrees $S_1, ..., S_p$ and $T_1, ..., T_q$, else return 0. If $p > q$ then return 0.

(2) If the number of T_j's with $V(T_j) \geqslant 2$ is smaller than the number of S_i's with $V(S_i) \geqslant 2$ then return 0.

(3) (Recursively) For each S_i such that $V(S_i) \geqslant 3$ (i.e., $D(\text{root}_{S_i}) \geqslant 3$) decide whether S_i is a rooted subtree of T_j with $V(T_j) \geqslant 3$ and form a $k \times l$ matrix M with $m_{ij}=1(0)$ if S_i is (not) a rooted subtree of T_j. k is the number of such S_i's and l is the number of such T_j's. Relabel the S_i's and T_j's if necessary.

(4) Apply to M an algorithm for finding a maximal matching in a bipartite graph. If there is no such matching for M then return 0 else return 1.

It has been shown by Verma (1988) and Verma and Reyner (1989) that given an algorithm for bipartite matching which requires at most $O(rs^u)$ operations, where $r \leqslant s$ are the sizes of the vertex sets and $u > 1$, the above algorithm requires at most $O(nm^u)$ operations.

We can easily extend the preprocessing step, outlined above, to GACML also. We need to collect at most two vertex labels for subtrees of size less than 3, and ensure that these are identical to the vertex labels of the matching subtree in the other term (except when the label in the pattern is a variable). It is also easy to extend the algorithm for subtree isomorphism to get an algorithm for GACML. In Step 1, we need to compare the vertex labels of the two roots. Also, we need to add some steps to handle the case of an uninterpreted operator, and the case when an operator is associative only. In the first case we need not apply a bipartite matching algorithm (because of the ordering of the arguments, which must be preserved), and in the second case we need an *ordered bipartite matching*, defined in the next section, which is easier to find than a maximum bipartite matching.

The commutative operator is treated just like an AC operator. Thus, from Theorem 10 and the fact that bipartite matching can be done in or $O(rs^{1.5})$ time ($r \leqslant s$), we have the following result.

COROLLARY 7. *GACML can be done in* $O(|s|\ |t|^{1.5})$ *time.*

We now give a linear time reduction from maximum bipartite matching to ACML.

THEOREM 16. *Maximum bipartite matching* $\leqslant_m^P$ *ACML.*

Proof. Given a bipartite graph $G = (V_1, V_2, E)$ with $|V_1| \leqslant |V_2|$ we construct two flattened linear terms s and t as follows. Let $V_1 = \{v_1, ..., v_k\}$ and $V_2 = \{v_{k+1}, ..., v_{k+l}\}$, where $k = |V_1|$ and $l = |V_2|$. We let $V = \{x_1, ..., x_k, y\}$ and $F = \{v_1, ..., v_k, f, g, a, b\}$, where f and g are binary AC operators, and a, b, and the v_i's are constants. For each vertex $v_i \in V_1$ we construct a term $s_i = f(v_i, x_i)$. For each vertex $v_j \in V_2$ with edges to vertices $v_{i_1}, ..., v_{i_m}$ (all distinct) we construct a term $t_j = f(v_{i_1}, ..., v_{i_m})$. If $v_j \in V_2$ has less than two edges we use the dummy constants a and b, as required, to construct the term t_j. Let $t = g(t_1, ..., t_l)$. Finally, if $k < l$, then let $s = g(s_1, ..., s_k, y)$, otherwise let $s = g(s_1, ..., s_k)$. It should be clear that s ac-matches t iff a matching of size $|V_1|$ exists in the bipartite graph G, and that s and t are flattened linear terms. Also, the time required for the reduction is linear in the size of G. ∎

An important consequence of this mutual reducibility is that our upper bound is tight, for any nontrivial improvement in the running time of one of the problems will improve that of the other.

4.1.4. *Linear Associative Matching*

Now we present an efficient $O(|s|\ |t| \log |s|)$ time sequential algorithm for AML that takes as input two trees s and t, and checks that s a-matches t. The sequential algorithm for AML and the space-efficient turing machine for AML, in Section 5.1, are based on the following characterization. Since our definitions of equational matching are based on logical consequence from an equational theory, we have to prove that such characterizations are correct. Fortunately, however, for this case it is not very difficult.

LEMMA 4 (AML Characterization). *A linear term P a-matches a term S iff one of the following holds:*

- *P is a variable.*
- *$P = f(P_1, ..., P_m)$ for some non-associative function symbol f of arity $m \geqslant 0$, $S = f(S_1, ..., S_m)$ and P_i a-matches S_i for $i \in I_m$.*

• *$P = f(P_1, ..., P_m)$ for some associative function symbol f of arity $m \geqslant 2$, $S = f(S_1, ..., S_n)$, $m \leqslant n$, and the ordered sequence $1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant m$ of all indices of non-variable terms from the P_i's, corresponds to an ordered sequence $1 \leqslant j_1 < j_2 < \cdots < j_k \leqslant n$ satisfying*

1. *$i_l \leqslant j_l$ for $l \in I_k$.*
2. *(boundary conditions) $i_1 = 1 \Rightarrow j_1 = 1$ and $i_k = m \Rightarrow j_k = n$.*
3. *$i_{l+1} - i_l \leqslant j_{l+1} - j_l$ and $i_{l+1} - i_l = 1 \Rightarrow j_{l+1} - j_l = 1$ for $l \in I_k$.*
4. *P_{i_l} a-matches S_{j_l} for $l \in I_k$.*

Proof. By induction on $|s|$ and $|t|$—straightforward. ∎

The above lemma motivates the following problem, which we call *ordered bipartite matching*. Abstractly, let $G = (V_1, V_2, E)$ be a bipartite graph, $V_1 = \{1, ..., m\}$, $V_2 = \{1, ..., n\}$, $m \leqslant n$, and $E \subseteq \{(i, j) \mid i \in V_1, j \in V_2\}$. Let $V \subseteq V_1$ be given. Then a matching M (a set of edges such that no two of them share a vertex) of size $|V|$ is called an ordered bipartite matching, it is satisfies the first three conditions of the above lemma, which we restate for clarity.

1. $(i, j) \in M \Rightarrow i \leqslant j$ for each $i \in V$.
2. $1 \in V \Rightarrow (1, 1) \in M$ and $m \in V \Rightarrow (m, n) \in M$.
3. For any $k \in I_{m-1}$, if vertices k and $k+1$ are in V and $(k, l) \in M$, then $(k+1, l+1) \in M$. Also, if $k, k+p \in V$ are such that the vertices $k+1, ..., k+p-1$ are not in V and $(k, l), (k+p, l+q) \in M$, then $p \leqslant q$.

The characterization given above suggests the following algorithm for AML.

ALGORITHM. Linear Associative Matching.

```
if s ∈ V then
  return true
else
  if root label of s = root label of t then
    Delete the roots of s and t to obtain
    rooted trees s_1, ..., s_m and t_1, ..., t_n
    case root label s of
      assoc:
        if m ≤ n then
          For each s_i ∉ V recursively decide whether s_i a-matches t_j for
          j ∈ {i, ..., n} and form an m × n matrix M with m_ij = 1 (0) if s_i
          a-matches (does not match) t_j
        else
          return false;
```

return true or false according to whether there is an ordered bipartite matching in the graph represented by M

not assoc:

Recursively determine, for $i \in I_m$, $match_i = s_i$ a-matches t_i;

return $\bigwedge_{i=1}^{m} match_i$

else return false

The correctness of this algorithm follows from Lemma 4.

THEOREM 17. *The time complexity of the above algorithm for AML is* $O(|s|\ |t| \log |s|)$.

Proof (Sketch). The idea of the proof is to show that given $G = (V_1, V_2, E)$ and $V \subseteq V_1$, an ordered bipartite matching can be found in $O(|V_1|\ |V_2|)$ time. Then using Theorem 9 we have the required result. Theorem 9 still applies because the proof of Theorem 9 makes no assumption about the bipartite matching algorithm, except for its complexity. Therefore, even though we are solving a slightly different (and easier) problem, the analysis is still applicable. To find an ordered bipartite matching in the stated time, we proceed as follows. Group the vertices in V into clusters, where each cluster is a maximal collection of consecutive vertices (recall that vertices in V_1 are numbered 1 through m). To each cluster we assign a number, which is the number of the lowest numbered vertex in the cluster. Now consider the clusters in ascending order and determine for each cluster $c = \{i, i+1, ..., i+k\}$ of size $k+1$, the first cluster of available vertices (i.e., those which have not been assigned to some previous cluster) from V_2, say $d = \{j, j+1, ..., j+k\}$ such that $i \leqslant j$, and $(i+l, j+l) \in E$ for all $l \in I_k$. If this is not possible for some cluster, then G has no ordered bipartite matching. Thus the conditions of the problem enable us to avoid any backtracking, and it is easy to see that the complexity of this algorithm is $O(|V_1|\ |V_2|)$. ∎

4.1.5. *Linear Commutative Matching*

We observe that the naive algorithm for linear CM, which treats the commutative operator just as an AC operator, takes only $O(|s|\ |t|)$ time, because a commutative operator can have only 2 arguments (hence the matching problem takes constant time). No graph matching is required for an uninterpreted operator.

4.2. *Parallel Complexity*

Designing parallel algorithms for AM, CM, and ACM with linear terms seems to be a formidable task. Hence we study the space complexity of these problems on the Turing machine model to see whether they are

parallelizable. We show that AML and CML are in NL. Thus from a well-known result of Borodin, we have AML, CML $\in NC^2$. We also show that GACML is mutually NC-reducible to bipartite matching. Thus the intriguing problem of the membership of bipartite matching in NC is now linked to existence of an NC algorithm for GACML. Since bipartite matching is known to be in RNC, we have shown that GACML is also in RNC. We first present a log-space nondeterministic turing machine accepting AML.

4.2.1. *Linear Associative Matching*

We assume that the two terms, s and t, are given as labeled ordered trees and that they are well-formed. It is not very difficult to design a deterministic Turing machine which uses only logarithmic space to check that the input is well-formed. The input is given on tape in the form of quintuples (u, v, x, y, z), where u and v are vertices, u is the parent of v, x and y are the labels of u and v, respectively, and z is the label of the edge from u to v. If u is labeled by an AC or commutative operator then $z = 0$. The roots of the two trees are given in the beginning of the description of each input tree. Also the two input trees are separated by some special symbol. Thus our input consists of two lists of the edges of s and t. The theorem also holds for the inverted and uninverted tree representations. It is only slightly more difficult to find the children of a node in the former representation, and the parent of a node in the latter. The details are messy but routine, and hence omitted.

THEOREM 18. *AML* $\in$ *NL.*

Proof (Sketch). From Lemma 4, we can design a nondeterministic turing machine M accepting AML. We give below a high level description of M as a RAM program.

```
temp1 := root(s); temp2 := root(t); state := descend;
repeat
  case state of
    descend:
      if label(temp1) ∈ V then
        state := ascend;
      else
        if label(temp1) = label(temp2) then
          case label(temp1) of
            assoc:
              if temp1 is being visited for the first time then
                verify degree constraints (see Lemma 4);
              find next non-variable child of temp1;
```

```
                if none then
                  state := ascend;
                else
                  guess next child of temp2 satisfying Lemma 4;
                  if not found then reject and halt
                  else update temp1 and temp2;
              non-assoc and not constant:
                find next non-variable child of temp1;
                if none then
                  state := ascend;
                else
                  find corresponding child of temp2;
                  update temp1 and temp2;
              constant: state := ascend;
            else reject and halt;
        ascend:
          find parent(temp1) and parent(temp2);
          if none then accept and halt
          else update temp1 and temp2 and state := descend;
forever;
```

The correctness of this algorithm follows from Lemma 4. To see that only logarithmic work space is required, observe that space is only needed for at most 6 registers to store the roots of current subterms being matched, temporary storage, and at most 4 counters for finding next children, etc. The Turing machine does not have to store the subterms being matched on its work tape; instead it uses the input tape as a "read-only stack." ▮

Combining Theorem 18 with Theorem 8, we have the following result.

COROLLARY 8. *AML is in uniform* NC^2.

4.2.2. *Linear Commutative Matching*

It is not very difficult to show that linear CM is in NL, since a nondeterministic Turing machine can guess (and then verify) the "corresponding" vertex in the other term for a child vertex, when the parent vertex is labeled by a commutative function symbol. As in the AML case the Turing machine need only store the root vertices of the current subterms being matched.

THEOREM 19. $CML \in NL$ *and hence also* NC^2.

4.2.3. *Linear Associative Commutative Matching*

First, we give a simple NC reduction from maximum bipartite matching to ACML. Thus an NC algorithm for ACML would guarantee an NC algorithm for maximum bipartite matching. No such algorithm exists. We then consider the case when all function symbols, except for nullary and unary ones, are associative as well as commutative. With this restriction, we give an NC reduction from ACML to rooted subtree isomorphism. Combining these two reductions with a result of Lingas and Karpinski (1986), we have the important result: restricted ACML $\equiv_{NC}$ subtree isomorphism $\equiv_{NC}$ bipartite matching. This also establishes membership of (restricted) ACML in RNC since bipartite matching has been shown to be in RNC (Mulmuley *et al.* 1987). Finally, we consider GACML, i.e., the most general AC matching problem. For this case, because of the complex interplay of various constraints we are not able to give a direct NC reduction to rooted subtree isomorphism. However, we are able to give an NC reduction from GACML to bipartite matching. This reduction is similar in some respects to the reduction given by Lingas and Karpinski (1986), but additional constraints make the reduction and its proof of correctness more involved.

THEOREM 20. *Maximum bipartite matching* $\leqslant^{NC}$ *ACML.*

Proof. The reduction of Theorem 16 can also be carried out with an NC^1 circuit with one oracle node for ACML. It is of some interest to note that the reduction can even be carried out by a polynomial-size constant-depth circuit! ∎

THEOREM 21. *Restricted ACML* $\leqslant^{NC}$ *rooted subtree isomorphism.*

Proof. Let s and t be the two rooted labeled trees representing the flattened linear terms, over the function symbols $F = \{f_1, ..., f_n\}$, to be AC matched. The idea is to get rid of the vertex labels from s and t to get two trees S and T, while ensuring two necessary and sufficient requirements for the reduction. The vertex set of s will be a subset of the vertex set of S. Similarly, the vertex set of t will be a subset of the vertex set of T. Thus, any root preserving isomorphism $\psi: S \to T$ has a restriction $\psi': s \to t$. We must constrain ψ in such a way that its restriction ψ' satisfies the following two conditions:

1. ψ' should preserve vertex labels, except when the label in s is a variable.

2. If v is any vertex in s labeled by an AC operator, v has no child labeled with a variable, and $\psi'(v) = u$, then the degree of v in s should equal the degree of u in t (recall Lemma 1).

To ensure these constraints we attach some gadgets to each vertex in the two trees s and t and strip the vertex labels to get S and T. Let $m = |s| + |t| + 1$. For each function symbol f_i we construct a gadget g_i which consists of a vertex r with $m + i$ vertices $u_1, ..., u_{m+i}$ as children, and vertex u_1 has $n + 1 - i$ children, $y_1, ..., y_{n+1-i}$. Now to every vertex in s and t labeled by function symbol f_i for some i, we attach gadget g_i by making r a new child of the vertex, to get trees S' and T'. Now for every vertex v_i in S' that is labeled by an AC operator and has no variable as a child, we find its degree d_i and attach gadget g_i to vertex y_1. Also to every vertex v_i in t, with degree $d_i > 0$ and labeled by f_j, attach gadget g_i to the vertex y_1 of gadget g_j (attached before to get T'). Finally remove all vertex labels to get S and T completing the reduction. The correctness of this reduction rests on the way the gadgets were chosen. Observe that any rooted isomorphism from S to a subtree of T must map gadgets onto gadgets. Once this is proven, showing that gadgets can be placed on one another only in such a way that the requirements are met is straightforward. It should be clear that entire reduction can be carried out by an NC^1 circuit. ∎

Finally, we consider GACML. To understand the difficulty of reducing GACML to rooted subtree isomorphism, recall that GACML problem includes every possible situation, i.e., operators which are associative only, operators which are commutative only, free operators, and AC operators. Now, consider the case of an associative operator which has some variables as children and which occurs more than once as the child of an AC operator. The interplay of (1) the freedom allowed by an AC operator, with (2) the constraints imposed by an associative operator, when combined with (3) the fact that the reduction must rely on "local" information, (as a priori, we have no information as to which vertex in the subject does this associative operator correspond) is very hard to capture in subtree isomorphism. However, we are able to reduce GACML to maximum bipartite matching.

It was shown in Benanav *et al.* (1985) that ACML is in P by a recursive reduction of ACML to maximum bipartite matching. This reduction can be extended in the obvious way to solve GACML. Unfortunately, however, the depth of this recursive reduction is proportional to the height of the pattern tree. To obtain an NC reduction we need to cut the size of the pattern repeatedly (to reduce the depth of the reduction). Our reduction is based on the notion of *corresponding paths.*

Let S and T be any two labeled rooted trees representing the flattened terms s and t, and let $P_1 = (v_1, ..., v_n)$ and $P_2 = (u_1, ..., u_m)$, $n, m > 0$, be two paths in S and T respectively starting from the roots v_1 and u_1. We say that paths P_1 and P_2 are *corresponding* iff the following conditions are satisfied:

1. $m = n$.

2. $label(v_i) = label(u_i)$ for all $i \in I_n$.

3. if $label(v_i)$ is neither associative nor commutative for any $i < n$, then $label((v_i, v_{i+1})) = label((u_i, u_{i+1}))$.

4. if $label(v_i)$ is associative but not commutative for any $i < n$, then $label((v_i, v_{i+1})) \leqslant label((u_i, u_{i+1}))$.

THEOREM 22. *$GACML \leqslant^{NC}$ maximum bipartite matching.*

Except for some preprocessing, the recursive reduction is carried out by procedure GACML given below. To avoid cluttering up the algorithm we have omitted the base cases for the recursive procedure, which occur when the pattern is either a variable or a constant. These details can be easily filled in by the reader. Array $D[1:n]$, where $n = |P| + |S|$, contains, for each vertex v, the number of descendants of vertex v in $D[v]$.

PROCEDURE. GACML(P, S, D)

```
if |P| > |S| then return 0
  else
    find a path L and its length |L| in P such that each
    vertex v on L satisfies D(v) > |P|/2;
    for i = 0 to |L| pardo v_i ← ith vertex of P;
    for each vertex u in S pardo
      find the path L_u from u to the root of S;
      if L and L_u are corresponding then
        let u_i denote the ith vertex of L_u, 0 ⩽ i ⩽ |L|
        for i = 0 to |L| pardo
          if not degree_constraints(v_i, u_i) then match(i) ← 0
          else
            case label(v_i) of
              AC, C:
                for all pairs (r_1, r_2) where label(r_1) is not a variable and
                r_1 is a child of v_i not on L and r_2 is a child of u_i not on
                L_u pardo M(r_1, r_2) ← GACML(Δ^P_{r_1}, Δ^S_{r_2}, D/Δ^P_{r_1});
                match(i) ← max_matching(M)
              A:
                for all pairs (r_1, r_2) where label(r_1) is not a variable and
                r_1 is a child of v_i not on P and r_2 is a child of u_i not on
                P_u pardo M(r_1, r_2) ← GACML(Δ^s_{r_1}, Δ^t_{r_2}, D/Δ^s_{r_1});
                match(i) ← if there is an ordered matching O in M and
                (O is compatible with the edge (v_{i+1}, u_{i+1}) or i = |L|)
                then 1 else 0
```

otherwise:
let $q_1, ..., q_k$ be all the children of v_i not on L
and $r_1, ..., r_k$ be the children of u_i not on L_u
for $i = 1$ to k **pardo** $M(q_i, r_i) \leftarrow GACML(\Delta^s_{q_i}, \Delta^t_{r_i}, D/\Delta^s_{q_i})$;
$match(i) \leftarrow \bigwedge_{j=1}^{k} M(q_j, r_j)$;
end{case};
parend;
$match(u) \leftarrow \bigwedge_{i=0}^{|L|} match(i)$;
else $match(u) \leftarrow 0$;
parend;
return $\bigvee_{u \in S} match(u)$;

The correctness of this procedure is justified by the following lemma. The lemma looks as though it is not symmetric. However, that is not really the case as the requirement "for one" is equivalent to "for each." The only reason for stating the lemma in this fashion is because it corresponds more closely to procedure GACML.

LEMMA 5 (GACML Characterization). *A flattened linear term s ac-matches a flattened term t only if (if) for each (for one) non-variable vertex $v \in s$ there exists a vertex $u \in t$ such that the paths $v_1, ..., v_n = v$ from v_1 the root of s and $u_1, ..., u_m = u$ from u_1 the root of t $(n, m > 0)$, are corresponding and for each $i \in I_n$*

- *$deg(v_i) \leqslant deg(u_i)$ with equality if $\forall v \in V\ v \notin arg(v_i)$.*
- *if $label(v_i)$ is either commutative or AC, then to all the non-variable children of v_i, $v_{i_1}, ..., v_{i_k}$ (all distinct) correspond k distinct children of $u_i, u_{i_1}, ..., u_{i_k}$ such that v_{i_j} ac-matches u_{i_j}, for $j \in I_k$.*
- *if $label(v_i)$ is associative only, then to the ordered sequence $v_{i_1}, ..., v_{i_k}$ $(1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant m)$ of all non-variable children of v_i, corresponds an ordered sequence $u_{j_1}, ..., u_{j_k}$ $(1 \leqslant j_1 < j_2 < \cdots < j_k \leqslant n)$ of children of u_i, satisfying*
  1. *$i_l \leqslant j_l$ for $l \in I_k$.*
  2. *(boundary conditions) $i_1 = 1 \Rightarrow j_1 = 1$ and $i_k = m \Rightarrow j_k = n$.*
  3. *$i_{l+1} - i_l \leqslant j_{l+1} - j_l$ and $i_{l+1} - i_l = 1 \Rightarrow j_{l+1} - j_l = 1$ for $l \in I_k$.*
  4. *v_{i_l} ac-matches u_{j_l} for $l \in I_k$.*
- *if $label(v_i)$ is neither associative nor commutative, then to the ordered sequence of all children $v_{i_1}, ..., v_{i_k}$ of v_i, corresponds the ordered sequence $u_{i_1}, ..., u_{i_k}$ of all children of u_i, such that v_{i_j} ac-matches u_{i_j} for all $j \in I_k$.*

Proof (Sketch). (Only if) Fix an s. The proof is by induction on the height of t.

Basis. t is any constant. If $s \in V$ there is nothing to prove. If s is a constant then s ac-matches $t \Rightarrow$ there exists σ such that $\sigma(s) = t \Rightarrow s = t$ since $\sigma(s) = s$ and we are done. If s is not a constant there is nothing to prove.

Induction Step. $t = f(t_1, ..., t_n)$, $n > 0$ and there are three cases to be considered.

Case 1. f is neither associative nor commutative. If $s \in V$ or $s = g(s_1, ..., s_m)$, $f \neq g$, $m > 0$, we are done. If $s = f(s_1, ..., s_n)$ and there exists σ such that $\overline{\sigma(s)} = t \Rightarrow f(\overline{\sigma(s_1)}, ..., \overline{\sigma(s_n)}) = f(t_1, ..., t_n) \Rightarrow \overline{\sigma(s_j)} = t_j$, $j \in I_k$. Thus the lemma is proved for the root and combining this with the induction hypothesis ($\overline{\sigma(s_i)} = t_i$) we are done.

Case 2. f is associative only. As before $s = f(s_1, ..., s_m)$ otherwise there is nothing to prove. Now for any σ, $|arg(\overline{\sigma(s)})| \geq |arg(s)|$ with equality for boolean terms (Lemma 2) therefore the constraints on degree for the root are proved. Since $s \in \bar{\mathcal{T}}$ none of the s_i's is of the form $f(...)$. Similarly none of the t_i's is of the form $f(...)$. Thus, if $i_1, ..., i_k$ are defined as in the lemma then $rem(\overline{\sigma(s_{i_j})}, f) = \overline{\sigma(s_{i_j})}$ for $j \in I_k$. Hence we can show that $\sigma(s) =_A t \Rightarrow \exists$ an ordered sequence $j_1, ..., j_k$ such that $\overline{\sigma(s_{i_l})} = t_{j_l}$ for $l \in I_k$. The stated conditions for this case are obvious since $|\sigma(v)| \geq 1$. Now apply the induction hypothesis and observe that the path to the root for each vertex in the s_i's is longer by one vertex (the root) and we have just proved the lemma for the root.

Case 3. f is either commutative or both associative and commutative and $s = f(s_1, ..., s_m)$. It can be shown that, σ such that $\sigma(s) =_C t$ or that $\sigma(s) =_{AC} t \Rightarrow \exists 1\text{-}1 \pi: I_m \to I_n$ such that s_i matches t_j, $i \in I_m$ and $j \in I_n$. Then we invoke the induction hypothesis.

(if) Let v and u be two vertices in s and t satisfying the conditions of the lemma. It is easy to show that if $v_1, ..., v_n = v$ and $u_1, ..., u_n = u$ are corresponding paths as in the lemma, then $\Delta^s_{v_i}$ ac-matches $\Delta^t_{u_i}$, for all i, which implies that s ac-matches t. ∎

The correctness of procedure GACML now follows from the above lemma and the following fact:

Fact 1. The path P chosen in procedure GACML does not end on a variable.

Now we show that procedure GACML can be implemented by an NC circuit with oracle gates for bipartite matching.

Let $n = |s| + |t|$. Clearly the recursion depth of procedure GACML is $\log n$ since it depends on $|s|$ and each time we halve the size of s for the recursive calls. Now observe that there is a unique path in P satisfying the given conditions. This path can be chosen on a CREW PRAM with a processors in $\log n$ time by standard techniques. Similarly the path for

TABLE 4

Table Summarizing Results in Section 4

Problem	Model	Result
2-occurrence AM	Sequential	NP-complete
2-occurrence CM	Sequential	NP-complete
2-occurrence ACM	Sequential	NP-complete
Linear AM	Sequential	$O(\|s\| \, \|t\| \log s)$
Linear CM	Sequential	$O(\|s\| \, \|t\|)$
Linear ACM	Sequential	$O(\|s\| \, \|t\|^{1.5})$
Boolean AM	Both	$O(\|s\| + \|t\|)$, NC^1
Boolean CM	Sequential	NP-complete
Boolean ACM	Sequential	NP-complete
Linear AM	Parallel	NC^2
Linear CM	Parallel	NC^2
Linear ACM	Both	$\equiv$ Bipartite Matching

each vertex in T can be obtained by n^2 processors in $\log n$ time on a CREW PRAM. By Stockmeyer and Vishkin (1984) this can be done by polynomial size (uniform) circuits of unbounded fan-in and $O(\log n)$ depth and hence by NC^2 circuits. Checking whether two paths are corresponding and whether one vertex has more sons than another are easily done by NC^1 circuits. Also the maximum number of son pairs (r_1, r_2) is at most n^2 and therefore the depth of the oracle gates is at most $2 \log n$. Therefore, except for the recursive calls and bippartite matching tests the procedure can be implemented by NC circuits. Finally, observe that procedure GACML makes at most n^2 *different* recursive calls. Hence the entries of the matrix M can be filled up in bottom-up fashion and thus the entire procedure can be implemented by NC circuits. Note that the values of array D have to be computed only once and this can be done by the Euler-tour technique of Tarjan and Vishkin (1985) using n processors in $O(\log n)$ time on an EREW PRAM. Hence it can also be done by uniform circuits of unbounded fan-in $O(\log n)$ depth and polynomial size, and hence by NC^2 circuits. This concludes the proof.

Results of this section are summarized in Table 4.

5. Conclusion

In this paper we first presented lower bounds for (uninterpreted) term matching with several representations on various models of parallel computation. These lower bounds have been shown to be tight by giving matching upper bounds for some representations and several models. Some interesting problems remain open. The time complexity of term matching

with trees, on the CRCW model, is still $O(\log n)$. Thus the lower bound here is slightly weak. We feel that a matching upper bound on trees would imply a sub-logarithmic algorithm for list ranking. No such algorithm is known.

We then investigated the complexity of interpreted term matching when some of the function symbols in a term are associative or commutative. We established several tight results on the sequential and parallel complexity of associative commutative matching and its variants. In some applications, it may be interesting to consider the case when the number of distinct variables is bounded. With this restriction all three problems, associative matching, commutative matching, and associative–commutative matching, have polynomial time algorithms. To see this, note that the total number of distinct substitutions for the variables in the pattern are polynomially bounded, and checking for each substitution that the two terms so obtained are identical modulo the ac-axioms can be done in polynomial time. We conclude by mentioning some open questions on the parallel complexity of these problems.

We can show that CML is in Dlogspace. However, whether this is the case for AML or whether AML is complete for Nlogspace is open. It would be surprising if the problem fell in the intermediate category. Finally, observe that our Turing machines were based on the unique parent property of trees. Hence the exact parallel complexity of AML, CML, and ACML for DAG's is also open.

Note added in proof. Recently, the first author has improved the sequential upper bounds for AML (to $O(|s|\ |t|)$) and GACML.

ACKNOWLEDGMENTS

The authors thank the two referees for their detailed comments and suggestions.

RECEIVED October 19, 1988; FINAL MANUSCRIPT RECEIVED November 21, 1990

REFERENCES

BACHMAIR, L., DERSHOWITZ, N., AND HSIANG, J. (1989), Completion without failure, *in* "Resolution of Equations in Algebraic Structures. Vol. II. Rewriting Techniques," pp. 1–30, Academic Press, New York.

BEAME, P., AND HASTAD, J. (1987), Optimal bounds for decision problems on the CRCW PRAM, *in* "Proceedings of the ACM Symposium on Theory of Computing," pp. 83–93.

BENANAV, D., KAPUR, D., AND NARENDRAN, P. (1985), Complexity of matching problems, *in* "Proceedings of the Conference on Rewriting Techniques and Applications," Vol. 202, pp. 417–429, Springer-Verlag, Berlin/New York; also in *J. Symbolic Comput.* **3** (1987), 203.

BORODIN, A. (1977), On relating time and space to size and depth, *SIAM J. Comput.* **6**, No. 4, 733.

CHRISTIAN, J. (1989), Fast Knuth–Bendix completion, summary, *in* "Proceedings of the Conference on Rewriting Techniques and Applications," pp. 551–555.

COLE, R., AND VISHKIN, U. (1986), Approximate and exact parallel scheduling with applications to list, tree and graph problems, *in* "Proceedings of the IEEE Conference on Foundations of Computer Science," pp. 478–491.

COOK, S., AND DWORK, C. (1982), Bounds on the time for parallel RAM's to compute simple functions, *in* "Proceedings of the ACM Symposium on Theory of Computing," pp. 231–233.

COOK, S. A. (1985), A taxonomy of problems with fast parallel algorithms, *Inform. Contr.* **64**, 2.

DWORK, C., KANELLAKIS, P., AND MITCHELL, J. C. (1984), On the sequential nature of unification, *J. Logic Programming* **1**, 35.

DWORK, C., KANELLAKIS, P., AND STOCKMEYER, L. (1986), Parallel algorithms for term matching, *in* "Eighth Conference on Automated Deduction."

FORTUNE, S., AND WYLLIE, J. (1978), Parallelism in random access machines, *in* "Proceedings of the ACM Symposium on Theory of Computing," pp. 114–118.

GAREY, M. R., AND JOHNSON, D. S. (1979), "Computers and Intractability: A Guide to the Theory of NP-Completeness," Freeman, San Francisco.

KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. (1985), Efficient parallel algorithms for graph problems, *in* "Proceedings of International Conference on Parallel Processing," pp. 180–185.

LINGAS, A., AND KARPINSKI, M. (1986), Subtree isomorphism and bipartite perfect matching are mutually NC-reducible, *in* "Research Report 856—CS," Universität Bonn.

MULMULEY, K., VAZIRANI, U. V., AND VAZIRANI, V. V. (1987), Matching is as easy as matrix inversion, *in* "Proceedings of Nineteenth Symposium on Theory of Computing," pp. 345–354.

O'DONNELL, M. J. (1985), "Equational Logic as a Programming Language," MIT Press, Cambridge, MA.

PATERSON, M. S., AND WEGMAN, M. N. (1978), Linear unification, *J. Comput. System Sci.* **16**, 158.

RAMESH, R., AND RAMAKRISHNAN, I. V. (1987), Optimal speedups for parallel pattern matching in trees, *in* "Proceedings of the Conference on Rewriting Techniques and Applications."

RAMESH, R., RAMAKRISHNAN, I. V., AND WARREN, D. S. (1990), Automata-driven indexing of prolog clauses, *in* "Proceedings of the ACM Symposium on Principles of Programming Languages."

RAMESH, R., VERMA, R. M., KRISHNAPRASAD, T., AND RAMAKRISHNAN, I. V. (1989), Term matching on parallel computers, *J. Logic Programming* **6**, 213.

REYNER, S. W. (1977), An analysis of a good algorithm for the subtree problem, *SIAM J. Comput.* **6**, 730.

SHILOACH, Y., AND VISHKIN, U. (1981), Finding the maximum, merging, and sorting in a parallel computation model, *J. Algorithms* **2**, 88.

STOCKMEYER, L., AND VISHKIN, U. (1984), Simulation of parallel random access machines by circuits, *SIAM J. Comput.* **13**, No. 2, 409.

TARJAN, R. E., AND VISHKIN, U. (1985), An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* **14**, No. 4, 862.

VERMA, R. M. (1988), "An Error in Reyner's—An Analysis of a Good Algorithm for the Subtree Problem," Technical Report 88/03, S.U.N.Y. at Stony Brook Computer Science Dept.

VERMA, R. M., KRISHNAPRASAD, T., AND RAMAKRISHNAN, I. V. (1986), An efficient parallel algorithm for term matching, *in* "Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science," Vol. 241, pp. 504–518, Springer-Verlag, Berlin/New York.

VERMA, R. M., AND REYNER, S. W. (1989), An analysis of a good algorithm for the subtree problem, corrected, *SIAM J. Comput.* **18**, No. 5, 906.