SIAM J. COMPUT.
Vol. 17, No. 4, August 1988

008

PARALLEL ALGORITHMS FOR TERM MATCHING*

CYNTHIA DWORK†, PARIS C. KANELLAKIS‡, AND LARRY STOCKMEYER†

Abstract. We present a randomized parallel algorithm for term matching. Let n be the number of nodes of the directed acyclic graphs (dags) representing the terms to be matched. Then our algorithm uses $O(\log^2 n)$ parallel time and $M(n)$ processors, where $M(n)$ is the complexity of $n \times n$ matrix multiplication. The randomized algorithm is of the Las Vegas type, that is, the answer is always correct, although with small probability the algorithm might fail to produce an answer. The number of processors is a significant improvement over previously known bounds. Under various syntactic restrictions on the form of the input dags, only $O(n^2)$ processors are required in order to achieve deterministic $O(\log^2 n)$ parallel time. Furthermore, we reduce directed graph reachability to term matching using constant parallel time and $O(n^2)$ processors. This is evidence that no deterministic algorithm can significantly beat the processor bound of our randomized algorithm. We also improve the P-completeness result of Dwork, Kanellakis, and Mitchell on the unification problem, showing that unification is P-complete even if both input terms are linear, i.e., no variable appears more than once in each term.

Key words. unification, term matching, parallel algorithms, logic programming

AMS(MOS) subject classification. 68Q

1. Introduction.

Unification of terms is an important step in resolution theorem proving [14], with applications to a variety of symbolic computation problems. In particular, unification is used in PROLOG interpreters [2], [7], type inference algorithms [10] and term-rewriting systems [6]. Informally, two symbolic terms s and t are unifiable if there exists a substitution of additional terms for variables in s and t such that under the substitution the two terms are syntactically identical. For example, the terms $f(x, x)$ and $f(g(y), g(g(z)))$ are unified by substituting $g(z)$ for y and $g(g(z))$ for x.

Unification was defined in 1964 by Robinson in his seminal paper "A Machine Oriented Logic Based on the Resolution Principle" [14]. Robinson's unification algorithm required time exponential in the size of the terms. The following years saw a sequence of improved unification algorithms, culminating in 1976 with the linear time algorithm of Paterson and Wegman [12]. A general interest in parallel computing, together with specific interest in parallelizing PROLOG, led Dwork, Kanellakis, and Mitchell to search for a fast (time polynomial in $\log n$) processor efficient (polynomially many processors) parallel unification algorithm [4]. Their results were negative: they proved that unification is complete for polynomial time, even if the input terms are represented as trees. A similar result was independently derived by Yasuura [18]. (The result in [18] is slightly weaker because it proves completeness for a more restricted class of inputs.) Thus, the existence of a fast, efficient parallel algorithm is *popularly unlikely*, in that it would contradict the popularly believed complexity theoretic conjecture that P, the class of problems solvable sequentially in polynomial time, is not

* Received by the editors July 28, 1986; accepted for publication (in revised form) June 24, 1987. This is a revised and expanded version of the paper "Parallel Algorithms for Term Matching," appearing in the Proceedings of the 8th International Conference on Automated Deduction, July 1986, Oxford, United Kingdom, Lecture Notes in Computer Science, Vol. 230, pp. 416–430, © 1986 by Springer-Verlag.

† IBM Almaden Research Center, Department K53/802, San Jose, California 95120.

‡ Computer Science Department, Brown University, Providence, Rhode Island 02912 and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. The research of this author was supported in part by an IBM Faculty Development Award and in part by National Science Foundation grant MCS-8210830.

contained in NC, the class of problems solvable in polylogarithmic time using polynomially many processors. However, Dwork et al. [4] found that term matching, a special case of unification in which one of the terms contains no variables, is in NC. Term s *matches* term t if there exists a substitution σ mapping variables in s to terms, such that $\sigma(s)$, the term obtained by replacing each occurrence of each variable x in s by $\sigma(x)$, is syntactically equal to t. Dwork et al. [4] obtained a matching algorithm requiring $O(\log^2 n)$ time and about $O(n^5)$ processors. Motivated by [4] some researchers interested in parallelizing PROLOG examined extant PROLOG programs to see whether in practice unification can be replaced by term matching. Preliminary results show that "often" the full power of general unification is not needed, and that term matching indeed suffices [9].

Let $\mathrm{NC}^k(f(n))$ be the class of problems solvable in time $O(\log^k n)$ using $f(n)$ processors on inputs of size n. Similarly, let $\mathrm{RNC}^k(f(n))$ be the class of problems solvable by a *randomized* algorithm in time $O(\log^k n)$ using $f(n)$ processors on inputs of size n. In defining these classes, our model of computation is the Concurrent-Read Exclusive-Write Parallel RAM (PRAM) [5], with word size $O(\log n)$ on inputs of size n.

The algorithm of [4] shows that term matching is in $\mathrm{NC}^2(M(n^2))$, where $M(m)$ is the number of arithmetic operations required for $m \times m$ matrix multiplication. Coppersmith and Winograd [3] show that $M(m) = O(m^{2.5})$, so the algorithm of [4] uses about n^5 processors. The current paper provides substantially improved upper bounds on processors for the term-matching problem at no asymptotic cost in running time. However, the new algorithm is randomized, in that the individual processors make random choices (flip coins). It is a Las Vegas algorithm: an answer is always correct, but there is some small probability (taken over the set of coin tosses) that an execution of the algorithm will fail to produce an answer. In that case the algorithm can be run again. Our approach will be to show how to test two terms for syntactic equivalence in $\mathrm{RNC}^2(M(n))$, where n is the total number of nodes in the dag representations of the two terms. This is the only randomized portion of the algorithm. We then show that term matching reduces to equivalence testing, and the reduction can be performed in $\mathrm{NC}^2(n^2)$. Since $M(n) \geqq n^2$ the principal result for term matching then follows as an easy corollary.

The remainder of the paper is organized as follows. Section 2 contains results on testing two terms for syntactic equivalence. In addition, this section contains some "evidence" that $M(n)$ is a lower bound on the number of processors needed for NC solutions to both term matching and equivalence, by showing that the directed acylic graph reachability problem reduces to testing for equivalence by an $\mathrm{NC}^0(n^2)$ reduction. Section 3.1 describes the reduction from term matching to equivalence testing. For certain special cases we can improve on the time and/or processor bounds obtained in the general case when both terms are represented by arbitrary dags. These results are described in § 3.2. Finally, § 4 strengthens the known P-completeness results for unification, showing that unification is P-complete even if both terms are linear (each variable appears at most once in each term) and are represented by trees, but where there can be sharing of variables. (In contrast, if there is no sharing of variables and one of the terms is linear, then the problem can be solved in NC as we show in § 3.3.) The proof of P-completeness of unification of linear terms is quite different from and more intricate than that of [4], and in a sense provides a strongest possible P-completeness result for a restricted form of unification.

2. Testing for equivalence of terms. A *term* is defined recursively as follows. A variable symbol is a term; if f is a k-ary function symbol, $k \geqq 0$ (0-ary function symbols correspond to constants), and $t_1, \cdots, t_k$ are terms, then $f(t_1, \cdots, t_k)$ is a term.

A *dag* is a directed acyclic graph. If G is a dag and u is a node of G, then the subdag *rooted at u* is the subdag consisting of all nodes and edges reachable from u. A term t can be represented by a labeled dag in a very natural way. If t is a constant or a variable, then the representation is just a single node labeled t. When $t = f(t_1, \cdots, t_k)$, $k \geq 1$, t can be represented by a dag consisting of a node labeled f with k outedges, labeled, respectively, 1 through k, such that the head of the edge labeled i is the root of a subdag representing t_i, for each $i = 1, \cdots, k$. Figure 1 shows some examples of labeled dags and their corresponding terms. Note that if the term contains a repeated subexpression then its corresponding dag is not unique; for example, a term $g(t, t)$ may be represented by using a single dag for both occurrences of t or by using a separate dag for each occurrence. A node u of a dag is a *root* if there are no edges directed into u, and u is a *leaf* if there are no edges directed out of u (each leaf must be labeled by either a constant or a variable). A term is *linear* if no variable appears more than once in the term. If the term t is linear and if G is any dag representation of t, then for each variable x occurring in t there is exactly one path in G from the root to a node labeled x. Let u and v be nodes of outdegree k, and let u_i (respectively, v_i) denote the head of the edge from u (respectively, v) with label i, $i = 1, \cdots, k$. Then we say u_i (v_i) is the *ith child of u* (respectively, of v), and we say u_i and v_i are *corresponding children* of u and v.

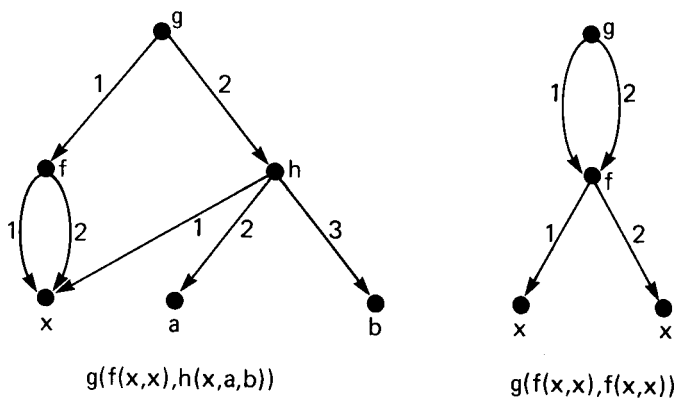

FIG. 1. *Some examples of term dags.*

Two rooted dags G and H are said to be *equivalent*, written $G \equiv H$, if they represent the same term. An instance of the equivalence problem is a triple (D, r_1, r_2), where D is a labeled dag with two roots r_1 and r_2. The two terms to be tested for equivalence are the terms represented by the subdags rooted at the two roots. (D could consist of two connected components, one with root r_1 and the other with root r_2. In general, we allow the dags rooted at r_1 and r_2 to share nodes since our equivalence algorithm handles this smoothly.) The time and processor complexities of our algorithms are expressed as functions of n, the size of an instance, defined to be the maximum of the number of nodes of D and the maximum outdegree of any node of D. Since in most applications the number of nodes will dominate the maximum outdegree, there is no harm in viewing n as the number of nodes. For cases where the maximum outdegree dominates, it is easy to express the complexities of our algorithms as functions of two separate parameters, the number of nodes and the maximum outdegree.

All logarithms in this paper are to the base 2.

2.1. A Las Vegas algorithm for testing equivalence of terms. Let $M(n)$ be an upper bound of the form cn^{ω}, for constants $c > 0$ and $\omega > 2$, on the number of arithmetic

operations in a straight-line algorithm which computes $n \times n$ matrix multiplication where the algorithm contains no divisions and where all scalars (i.e., numerical constants) are integers. The Coppersmith-Winograd algorithm [3] is of this form, so $M(n) = O(n^{2.5})$.

Consider an instance (D', r_1, r_2) of the equivalence problem and let n be its size. Let G' and H' be the dags rooted at r_1 and r_2, respectively. Since two terms are equal if and only if they are equal when we substitute constants for variables, we may assume that D' contains no variables. This will simplify matters in this section since we will want to use "variable" to refer to polynomial variables rather than term variables. Briefly, our approach is to represent G' and H' by multivariate polynomials $P_{G'}$ and $P_{H'}$ in such a way that G' and H' are equivalent if and only if the corresponding polynomials are equivalent. We then use a randomized algorithm of Schwartz [15] to check the equivalence of the polynomials. Since Schwartz's result deals with sequential algorithms, we have some additional work to prove that the algorithm can be modified to run in $\mathrm{RNC}^2(M(n))$.

In defining the polynomials, it is useful to first perform some modifications to the dag as follows. Given a labeled dag D', we first add a new node z to D', and add edges from every node in D' to the new node z, making z the unique leaf of the resulting dag, call it D. Each new edge (v, z) is D labeled with the outdegree of v in D. This process of adding a new leaf, connecting it to all nodes in the original dag, and labeling the new edges, is called *preparing* the dag, and D is said to be *prepared.* We view D as a term dag with the single constant symbol z; the arity of each function symbol labeling a node of D (except the new node) is one more than the arity of the function symbol labeling the corresponding node of D'. If G and H are the prepared dags rooted at r_1 and r_2, respectively, in D then it is obvious that G and H are equivalent if and only if G' and H' are equivalent. Therefore, we shall work with prepared dags in the remainder of this section.

Given a labeled prepared dag G, we define the corresponding polynomial P_G as follows. Our intention is to assign variables to edges; in § 2.1, "variable" means a variable of some polynomial P_G. Each path from the root to the leaf will then yield a monomial defined as the product of the variables assigned to the edges along the path. The final polynomial will then be the sum over all paths p from root to leaf of the monomial corresponding to p. We will show that two dags are equivalent if and only if their corresponding polynomials are equivalent. Reducing equivalence testing for term dags, which is in NC, to testing equivalence of polynomials, which is not even known to be in P, is not obviously progress. However, we will then apply the algorithm of Schwartz, modified to run in RNC, testing equivalence of the polynomials at a randomly selected point. A negative answer to this test is a proof of inequivalence. As we will see, a positive answer can sometimes be turned into a proof of equivalence. Sometimes a positive result will be inconclusive, in which case the algorithm can be run again.

We now describe the selection of the variables. Consider an arbitrary path from the root to the leaf. In selecting the variables corresponding to the edges of the path, care must be taken that the ordering information is not lost. In other words, given the monomial corresponding to the path we must be able to reconstruct the path. To this end, for each node v we compute the number of paths from v to z in G. Note that since G is prepared, if v is a proper ancestor of u then the number of paths from v to z exceeds the number from u to z; this fact will play an important role in the proof of Lemma 1.

VARIABLE NAMING RULE. Let v be a node of G labeled with the k-ary function

symbol f, and let m be the number of paths from v to the leaf z. Then for each $j \in \{1, \cdots, k\}$, f_j^m is the variable assigned to the edge from v labeled j.

For each path p directed from the root of G to z, let the monomial $\mu(p)$ be the product of all the variables corresponding to the edges of p. Then $P_G = \sum_p \mu(p)$.

LEMMA 1. *Two prepared term dags G and H are equivalent if and only if P_G and P_H are equivalent polynomials.*

Proof. We begin with the if direction. Let the *degree* of a polynomial denote the maximum number of variables in any monomial of the polynomial. Then $P_G \equiv P_H$ implies that the two polynomials are of the same degree. The proof proceeds by induction on k, the degree of the polynomials.

$k = 1$. In this case the two polynomials each consist of a single monomial (this is because of the way we defined a prepared dag). Thus, for some unary function symbols g and h we have $P_G = g_1^1$ and $P_H = h_1^1$. Clearly, these polynomials can be equivalent if and only if $g_1^1 = h_1^1$, so the symbols g and h must be identical.

$k > 1$. We assume the result inductively for polynomials of degree less than k and prove it for k. Let g (respectively, h) be the label of the root of G (respectively, H) and let i (respectively, j) be the number of paths from the root of G (respectively, H) to the leaf z. Write $P_G = \sum_{x=1}^{a} g_x^i t_x$ and $P_H = \sum_{x=1}^{b} h_x^j s_x$, where the superscripts in the t_x and s_x are less than i and j, respectively. Then $i = j$, as otherwise the maximum superscripts of the two sets of variables differ. Thus, none of the t_x contain any variable name of the form h_y^i and similarly, no s_x contains any g_y^i. Because the only function symbol superscripted with i in P_H is h, it follows that $g = h$, whence $a = b$. Rewriting P_H with g replacing h yields $P_H = \sum_{x=1}^{a} g_x^i s_x$. For any $x \in \{1, \cdots, a\}$, setting g_y^i to 0 for all $y \neq x$ yields $g_x^i t_x = g_x^i s_x$, whence $t_x = s_x$. Since the degree of t_x is strictly less than k, we see by the inductive hypothesis that the term represented by t_x is equivalent to the term represented by s_x. Let r_1 and r_2 be the roots of G and H, respectively. Since $g = h$ both roots are labeled with the same function symbol. Further, $t_x = s_x$ for all x, so the corresponding arguments of g are the same in the two dags. Thus, the two dags are equivalent.

We now prove that if the two prepared dags are equivalent then the corresponding polynomials are equivalent. Because the height of a dag corresponds to the maximum depth of nesting of parentheses in the term it represents, if $G \equiv H$ then the two dags are of the same height. The proof proceeds by induction on k, the height of the dags. We strengthen the induction hypothesis, showing that if two dags are equivalent then the number of paths from the root to the leaf is the same in the two dags.

$k = 1$. All prepared dags of height 1 contain a unique edge, from root to leaf, so the corresponding terms are just 1-ary function symbols. Thus, $G \equiv H$ implies the two dags are identical, whence they give rise to the same polynomial.

$k > 1$. We assume the result inductively for dags of height less than k and prove it for k. Let r_1 and r_2 be the roots of G and H, respectively. If $G \equiv H$ then the two roots are labeled with the same function symbol, say g, whence both roots have the same outdegree, say a. Further, because the dags are equivalent, for each $i \in \{1, \cdots, a\}$, the subdag rooted at the ith child of r_1 is equivalent to the subdag rooted at the ith child of r_2. Since these subdags are of height strictly less than k we see by the inductive hypothesis the polynomials corresponding to the subdags rooted at the ith children of the roots are equivalent, and the number of paths from the ith child of r_1 to the leaf is equal to the number of paths from the ith child of r_2 to the leaf, for all i. Thus, the total number of root-leaf paths is the same in the two dags. Let m denote this number. Let P_i denote the polynomial corresponding to the subdag rooted at the ith child of either root. Then $P_G = g_1^m P_1 + \cdots + g_a^m P_a$, and this is precisely P_H. □

Remarks. (1) If the variables were not superscripted with the path numbers Lemma 1 would not hold. This is because the variables f_1 and f_2 commute, i.e., $f_1 f_2 = f_2 f_1$, but the directed paths labeled $f_1 f_2$ and $f_2 f_1$ are not the same in a dag.

(2) Two very natural approaches to handling the commutativity problem do *not* work. Affixing $c \times c$ matrices to the edges, for some constant c, cannot work. This follows from a theorem of Amitsur and Levitzki [1] which states that there are polynomials $Q_1(x_1, \cdots, x_{2c})$ and $Q_2(x_1, \cdots, x_{2c})$ with zero-one coefficients, involving noncommuting variables $x_1, \cdots, x_{2c}$, such that Q_1 and Q_2 are not equivalent in general, but Q_1 and Q_2 are equivalent over the ring of $c \times c$ matrices. A second approach, computing for each node v the maximum distance from v to z (instead of the number of paths) requires a special kind of matrix multiplication, in which the inner operation is + and the outer operation is max. It is not known how to compute the "product" of two $n \times n$ matrices in $M(n)$ steps using this definition of multiplication.

In order to use Lemma 1 to test for equivalence of term dags, we must address several issues.

The number of paths from a node to the leaf may be as large as n^{n-1}, even though the outdegree of each node is bounded by n. Since the word size of our parallel random access machine (PRAM) is at most logarithmic in n we cannot actually compute these numbers. Similarly, we cannot evaluate the polynomials P_G and P_H, as for many possible choices of values for the variables the values of the polynomials will be too large to handle. We use modular arithmetic to handle these problems, performing all arithmetic modulo random primes, which in turn raises the question of how to obtain a random prime.

Testing equivalence of multivariate polynomials is not even known to be solvable in polynomial time, let alone in NC, even when we can actually evaluate the polynomials, much less when we cannot. For this we resort to a modification of Schwartz's randomized algorithm.

The Schwartz algorithm is Monte Carlo, in that an answer of "inequivalent" is always correct, but an answer of "equivalent" is correct only with high probability. Thus, the final issue is that of deriving a proof of equivalence when Schwartz's algorithm tells us two polynomials are equivalent. If the polynomials are arbitrary this problem is open. Our polynomials are not arbitrary; they are constructed from term dags. This can sometimes be exploited to obtain a proof of equivalence.

LEMMA 2. *For each constant k, there is an* $\mathrm{RNC}^2(\log^3 n)$ *algorithm that, on input n, with probability at least $1 - n^{-k}$ produces a random prime $q \in I = \{2, \cdots, n^k\}$.*

Proof. Our approach is to choose r random numbers, $q_1, \cdots, q_r \in I$ where $r = d \log^2 n$ for an appropriate constant d. Each q_i is tested for primality $(k+2) \log n$ times in parallel using Rabin's randomized test [13]. Of those q_i having passed all tests one is selected at random. Rabin's algorithm runs in time $O(\log^2 n)$ on inputs from I. When given a prime, the algorithm always answers "prime"; when given a composite, it answers "composite" with probability at least $\frac{1}{2}$.

There are two ways in which the algorithm could fail to produce a prime. It may be that none of the q_i are prime. The Prime Number Theorem implies that for some constant c, independent of n, the probability that all r are composite is at most $(1 - c/\log n)^r$, which is at most $n^{-k}/2$ if $r = d \log^2 n$, for some constant d independent of n. If at least one q_i is actually prime, it is still possible that a composite will be chosen. However, since Rabin's algorithm errs with probability at most $\frac{1}{2}$, the probability that a composite passes all $(k+2) \log n$ tests is at most n^{-k-2}. Thus, the probability that even one composite passes all tests is at most nn^{-k-2}, which is less than $n^{-k}/2$ for sufficiently large n. □

The first source of error in our randomized protocol for testing equivalence of term dags is in the selection of q, because our algorithm may fail to produce a prime. However, even if a prime q is produced, performing arithmetic modulo q introduces a second source of error, as distinct numbers r and s may be congruent mod q. Thus, when computing the superscripts for the variables mod q, two variables assigned to edges from nodes with differing numbers of paths to the leaf may receive the same superscript. We must therefore choose q from a range sufficiently large as to make this event unlikely.

DEFINITION. Integer m is *bad* for the pair of integers (r, s) if $r \neq s$ but r and s are congruent mod m.

LEMMA 3. *For every pair of distinct numbers $r, s \leqq n^n$ there are at most $n \log n$ bad primes.*

Proof. Let $\{p_1, \cdots, p_x\}$ be the set of primes bad for (r, s). Then the product $\pi = p_1 p_2 \cdots p_x$ is also bad for (r, s). Since $r, s \leqq n^n$ it must be that $\pi \leqq n^n$ (otherwise the two numbers could not be congruent mod π). On the other hand, since each $p_i \geqq 2$ we have $2^x \leqq \pi$. Thus, $2^x \leqq \pi \leqq n^n$, whence $x \leqq n \log n$, as was to be shown. □

Let us say a prime q is *bad* for a prepared dag if there exist two nodes u and v in the dag such that r and s are the number of paths to the leaf from u and v, respectively, $r \neq s$, and q is bad for r, s.

COROLLARY 4. *Let D be a prepared dag with n nodes in which each node has outdegree at most n. Then a random prime q drawn from $[2, n^k]$ is bad for D with probability at most $O(n^{3-k} \log^2 n)$ for any fixed $k \geqq 3$.*

Proof. There are at most n^2 pairs of nodes, and for each pair there are at most $n \log n$ bad primes, so there are at most $n^3 \log n$ primes bad for D. By the Prime Number Theorem there are $\Omega(n^k/(k \log n))$ primes in $[2, n^k]$. Thus, if a prime is selected uniformly at random from this range, then the probability that the chosen prime is bad for D is at most

$$O\left(n^2 \log n \frac{k \log n}{n^k}\right) = O(n^{3-k} \log^2 n). \qquad \square$$

Let Q be a polynomial in t variables with integer coefficients. An *assignment* for Q is a $(t+1)$-tuple of integers $A = (i_1, \cdots, i_t, p)$. We define

$$Q(A) = Q(i_1, \cdots, i_t) \qquad (\text{mod } p).$$

An assignment A is a *modular zero* of Q if $Q(A) = 0$. We let maxv (Q, m) denote the maximum value attained by Q over the rectangle in which the absolute value of each variable is bounded by m.

In order to bound the probability of error in testing the polynomials for equality, we appeal to the following theorem due to Schwartz [15].

THEOREM (Schwartz). *Let $2m + 1 \geqq c \cdot \deg(Q)$, let I be the set of integers of absolute value $\leqq m$, let J be a set of primes, and suppose that the product of the $c^{-1}|J| + 1$ smallest primes in J exceeds* maxv (Q, m). *Then if Q is not identically equal to zero, the number of elements of $I^t \times J$ which are modular zeros of Q is at most $2c^{-1}|I|^t|J|$.*

The Q we will be considering is $P_G - P_H$, which is of degree at most n (the number of nodes in the union of the two dags), and contains $t \leqq n^2$ variables. Suppose we wish to bound the probability of choosing a modular zero by n^{-k}. We apply Schwartz's Theorem as follows. Let $b \geqq k + 1$ and $m = n^b$. Let J be the set of primes in $[2, n^{2b}]$, and $I = [-n^b, n^b]$, where J is the set from which we select our random prime p and I is the interval from which each of our t variables is chosen. Each term in Q is the

product of at most n variables, and Q contains at most n^n terms, so

$$\operatorname{maxv}(Q, n^b) \leqq n^{bn} n^n.$$

To satisfy the conditions of the theorem we take $c = (2n^b + 1)/n$. Estimating the number of primes in J to be $dn^{2b}/2b \log n$, for some constant d, the condition that the product of the $c^{-1}|J|+1$ smallest primes exceeds $\operatorname{maxv}(Q, m)$ is clearly satisfied if

$$\frac{dn^{2b+1}}{(2n^b+1)2b \log n} \geqq n(b+1) \log n$$

(because each prime is no smaller than 2). This reduces to

$$dn^{2b} \geqq 2b(b+1)(2n^b+1)(\log n)^2,$$

which is clearly true for all sufficiently large b, assuming $n \geqq 2$. We therefore have the following corollary.

COROLLARY 5. *Let Q be a polynomial of degree $n \geqq 2$ such that $Q \not\equiv 0$. For any fixed k there is a constant b, depending only on k, such that by choosing a random assignment A for Q by choosing a random prime in the interval $[2, n^{2b}]$ and randomly selecting values for the variables in Q from $I = [-n^b, n^b]$, $Pr[Q(A) = 0] \leqq n^{-k}$.* □

When our polynomial Q is formed from two term dags it is obvious how we obtain a proof of inequality ($Q(A) \neq 0$), while if $Q(A) = 0$ we may have hit a modular zero at A. However, in some cases we can actually obtain a proof of equivalence. In part this is due to the fact that when we evaluate the polynomials corresponding to the two roots of our dags, we simultaneously evaluate all n polynomials corresponding to the subdags rooted at each of the n nodes, just as when testing directed graph reachability by computing the transitive closure of the adjacency matrix we obtain reachability information for all pairs of points.

Let D be a prepared dag with two roots, and let G and H be the subdags rooted at these roots. Choose variables for the edges according to the Variable Naming Rule. For each node v, let P_v be the polynomial induced by the subdag rooted at v. By convention, P_z is the identically zero polynomial where z is the unique leaf added when the dag was prepared. Running our parallelized version of Schwartz's algorithm with a particular assignment A to the variables induces an equivalence relation on the nodes of D, where distinct nodes x and y are in the same class if and only if the algorithm tells us $P_x(A) = P_y(A)$. Let us denote this relation by $x \equiv_A y$. For each pair of $\equiv_A$ nodes we check two things. First, we verify that both nodes are labeled with the same function symbol. Second, we check that the corresponding children of each pair of $\equiv_A$ nodes are also $\equiv_A$. We claim that if both these tests are passed for all pairs of $\equiv_A$ nodes then the subdags rooted at x and y are equivalent.

LEMMA 6. *Let D and A be as above. Let x and y be nodes of D satisfying $x \equiv_A y$. If for all u, v such that $u \equiv_A v$ and u and v both belong to the union of the subdags induced by x and y, it is the case that the labels of u and v agree and each pair of corresponding children of u and v are $\equiv_A$, then the subdags rooted at x and y are equivalent.*

Proof. The proof is by induction on k, the length of the longest path from x to the leaf z.

$k = 0$. In this case $x = z$, which has the label z. If the conditions of the lemma are satisfied then y has label z, so $y = z$ as well.

$k > 0$. Assume the result inductively for $k-1$, and assume the conditions of the lemma hold. Then the labels of x and y agree. Further, corresponding children of x and y are $\equiv_A$. Thus, by induction, the subdags rooted at their corresponding children are equivalent. It follows immediately that the subdags rooted at x and y are equivalent. □

Lemma 6 says it is sometimes possible to prove equivalence of polynomials generated from term dags. In particular, this can be done when for every pair of $\equiv_A$ nodes, the subdags rooted at the two nodes are actually equivalent. It is possible that two nodes labeled with different function symbols are "equivalent" under $\equiv_A$. In this case we can prove nothing about the ancestors of these nodes and we must run the algorithm again.

THEOREM 7. *The problem of testing equivalence of two term dags can be solved by a Las Vegas algorithm in* $\mathrm{RNC}^2(M(n))$. *For any fixed k the probability that no answer is produced can be bounded above by* n^{-k}.

Proof. The algorithm is as follows. The algorithm involves an integer parameter k' which is chosen depending on k. Let (D', r_1, r_2) be an instance of the equivalence problem, consisting of a dag D' and two roots.

1. Prepare D' by adding a new leaf z, adding an edge from each node to z, and labeling the added edges. Let D denote the resulting dag.
2. Choose a random prime $q \in [2, n^{k'}]$.
3. For each node v compute the number (mod q) of paths from v to z.
4. For each edge e compute the name of the associated variable according to the Variable Naming Rule.
5. Sort the variable names and remove duplicates from the sorted list.
6. Choose an assignment A by choosing random values from the range $[-n^{k'}, n^{k'}]$ for the variables in the list produced at the previous step and by choosing a random prime $p \in [2, n^{2k'}]$.
7. For each node v in D evaluate P_v, the polynomial corresponding to the subdag rooted at v, at the point A chosen in step 6. If $r_1 \not\equiv_A r_2$ then output "inequivalent" and halt.
8. If $r_1 \equiv_A r_2$ then try to prove equivalence by the method of Lemma 6. If successful, output "equivalent," else output "?".

We now describe steps 3, 7, and 8 in more detail.

For simplicity, let n denote the number of nodes in the prepared dag D (the original dag plus the new leaf). To perform step 3, we define an $n \times n$ matrix E whose ij entry is the number of edges from i to j (mod q). In other words, E_{ij} is the number (mod q) of paths of length 1 from i to j (diagonal entries are 0). In general, $(E^k)_{ij}$ is the number (mod q) of paths of length k from i to j. Since we are interested in all paths of all lengths from each node to z, we compute the sum of all powers of E from E^0 to E^{m-1} where $n \leqq m < 2n$ and m is a power of 2. To do this we use the well-known identity

$$I + E + E^2 + E^3 + \cdots + E^{m-1} = (I+E)(I+E^2)(I+E^4)\cdots(I+E^{m/2}).$$

Multiplication of $m \times m$ matrices can be computed in $\mathrm{NC}^1(M(n))$ (see, e.g., Pan and Reif [11, Appendix A]). Therefore, the powers of E needed to compute the rhs can be computed in $\mathrm{NC}^2(M(n))$ by repeated squaring. All arithmetic is done mod q. Since the rhs contains at most $\log n$ terms the product can also be computed in $\mathrm{NC}^2(M(n))$ once the powers of E are computed. Let $E^* = \sum_{i=0}^{m-1} E^i$ (mod q). Then for each node v, E^*_{vz} is the number (mod q) of paths from v to z.

The evaluations of the polynomials P_v proceed in a similar fashion. Thus, to perform step 7 we define a matrix B whose ij entry is the sum of the values chosen for the variables assigned to the edges from i to j. Viewed differently, B_{ij} is the value of the polynomial corresponding to the subdag containing paths of length 1 from i to j. In general, $(B^k)_{ij}$ is the value of the polynomial which is the sum of the monomials

corresponding to all paths of length k from i to j. As in the case of the computation of path numbers, we are interested in $B^* = I + B + B^2 + B^3 + \cdots + B^{m-1}$. This is computed as described above, all arithmetic being performed mod p. Then $B^*_{vz} = P_v(A)$. For any two nodes x and y we have

$$x \equiv_A y \Leftrightarrow B^*_{xz} = B^*_{yz}.$$

To perform step 8, the verification of equivalence, we proceed as follows. For all nodes x and y such that $x \equiv_A y$ we check that the labels of x and y are equal (if they are not then we can prove nothing, and the algorithm outputs "?"). Given that they are equal, we wish to check that corresponding children of $\equiv_A$ nodes are themselves $\equiv_A$. Let $e = (u, v)$ be an edge with label i. Corresponding to e there is a triple $(B^*_{uz}, i, B^*_{vz}) = (P_u(A), i, P_v(A))$. The set of triples corresponding to all the edges of the prepared dag are sorted lexicographically. We then examine the sorted list for a pair of adjacent triples whose first and second components match but whose third components differ. If no such pair exists the algorithm outputs "equivalent," else it outputs "?."

This completes the description of the algorithm. It remains to prove correctness when the random choices are good and to analyze the probability of making a bad random choice. Let G and H be the subdags rooted at r_1 and r_2, respectively.

By Lemma 1, $G \equiv H$ implies $P_G \equiv P_H$, and in the absence of a bad random choice in step 2, $G \not\equiv H$ implies $P_G \not\equiv P_H$. If $P_G \not\equiv P_H$, then in the absence of a bad choice for the assignment A at step 6, $r_1 \not\equiv_A r_2$, so we have a proof that $G \not\equiv H$. If $P_G \equiv P_H$, then, in the absence of a bad choice for A, for all pairs of nodes u, v such that $u \equiv_A v$ it is the case that the subdags rooted at u and v represent the same terms, in which case we have a proof of equivalence by Lemma 6.

This proves correctness in the absence of bad choices. We now bound the probability of making a bad choice.

As shown in Lemma 2 and Corollary 4, we can bound the probability of a bad choice in step 2 by n^{-k} for any fixed k. We now examine the probability of obtaining a "?" output given that no bad choice was made in step 2.

A "?" will be produced if for some pair of nodes u and v, the subdags rooted at the nodes are inequivalent but $u \equiv_A v$. By Lemma 2 and Corollaries 4 and 5 the probability of this occurring for a particular pair of nodes can be bounded above by n^{-k} for any fixed k. As there are only n^2 pairs of nodes, the probability of this event occurring for any pair of nodes can be similarly bounded.

This completes the proof of Theorem 7. □

2.2. Directed reachability reduces to equivalence testing. The *dag reachability* problem is: Given a dag D and two distinguished nodes s and t of D, does there exist a path from s to t? The size of the instance is the number of nodes in D. While directed reachability is known to be complete for NSPACE($\log n$) with respect to logspace reductions, and therefore to be in NC, little is known about the number of processors needed to solve this problem in polylog time. In fact, all known NC or RNC algorithms compute the transitive closure of the adjacency matrix for D by repeated squaring and therefore use $M(n)$ processors (to within logarithmic factors). Thus, while reducing directed reachability to term equivalence does not yield a lower bound on the number of processors needed to test for equivalence in NC, it does provide some "evidence" that $M(n)$, the processor bound obtained in Theorem 7, cannot be significantly improved.

THEOREM 8. *The directed acyclic graph reachability problem is* $\mathrm{NC}^0(n^2)$ *reducible to the equivalence testing problem.*

Proof. Given a dag D_1 with distinguished nodes s_1 and t_1 we will construct a pair of term dags E_1 and E_2 which will be equivalent if and only if there is no path from s_1 to t_1 in D_1. The construction is illustrated in Fig. 2.

Without loss of generality we assume D_1 has a unique root (if this is not the case then create a new root with edges to each of the original roots). We begin by turning D_1 into a term dag. For each k, $0 \leqq k \leqq n$, and for each node v of D_1 with outdegree k, label v with the function symbol f^k. Label the outedges from 1 to k in arbitrary order.

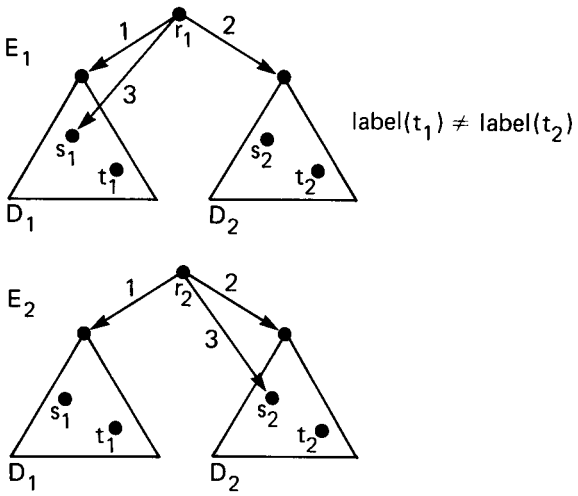

FIG. 2. *The dags constructed in the proof of Theorem* 8.

Create a copy D_2 of D_1, identical to D_1 but with t_2, the D_2 copy of t_1, labeled with a new function symbol g. Let s_2 be the D_2 copy of s_1. There is a path from s_1 to t_1 if and only if s_1 and s_2 are the roots of inequivalent subdags. However, we are not done, since the s nodes are internal, and we have been assuming that an equivalence algorithm takes as input two roots and determines whether the dags rooted at these roots represent equivalent terms.

We next create a new dag E_1 composed of D_1 and D_2 by creating a new root r_1 and making the roots of D_1 and D_2 first and second children, respectively, of r_1. Let s_1 be the third child of r_1. The edges from r_1 to its children are labeled accordingly, and r_1 is labeled with function symbol f^3.

Finally, we create a copy E_2 of E_1, identical to E_1 but with the 3-edge of r_2, the root of E_2, pointing to the E_2 copy of s_2 (instead of to the copy of s_1). The subdags rooted at the first and second children of the two roots are identical by construction. Thus, the two dags are equivalent if and only if the subdags rooted at the third children are equivalent. As observed above, this holds if and only if there is no path from s_1 to t_1 in D_1.

If D_1 contains n nodes then the four copies can be constructed in $O(1)$ time using n^2 processors (one per edge). The two new roots and six additional edges are added in constant time as well. □

3. Reducing term matching to equivalence testing. In § 2.1, we showed that two term dags can be tested for equivalence in $\mathrm{RNC}^2(M(n))$. We now show that these bounds apply to the term-matching problem as well. Recall that term s *matches* term t if and only if there exists a σ mapping variables to terms such that $\sigma(s) = t$. Note

that matching is not a symmetric relation. An instance of the term-matching problem is a pair of disjoint labeled term dags representing the terms s and t to be matched. The size of the instance is the number of nodes in the union of the two dags. Although our matching algorithm requires the two dags to be disjoint, nondisjoint instances can be transformed to disjoint instances by performing a reachability computation from the two roots; as in the proof of Theorem 7, this can be done in $\mathrm{NC}^2(M(n))$. In particular, disjointness is not needed in Corollary 10. Both term matching and equivalence testing are special cases of the unification problem. In the term-matching problem one of the terms (the second one) is considered *constant*: σ is not applied to the second term. In the equivalence problem, both terms are considered constant.

In § 3.1, we describe our results for general dags. Special cases are discussed in § 3.2.

3.1. Term matching on general dags. In this section we will prove the following theorem.

THEOREM 9. *Term matching reduces to testing for equivalence in* $\mathrm{NC}^2(n^2)$.

From Theorems 7 and 9 we obtain our main result.

COROLLARY 10. *Term matching can be solved by a Las Vegas algorithm in* $\mathrm{RNC}^2(M(n))$.

As mentioned in the Introduction we also have some specialized algorithms for term matching which depend on the form of the constant term. However, the general outline of the algorithm is the same in all cases. For ease of exposition we break the algorithm into four main steps. Since the first term, henceforth s, will always be represented by an arbitrary dag G, we may assume without loss of generality that each variable of s is the label of exactly one node of G (multiple copies of x can be merged into one copy).

Steps 1–4 below outline all our matching algorithms. Let G and H denote the dags representing s and t, respectively.

Match (G, H).

1. A processor is assigned to each node of G. A spanning tree T of G is formed by having the processor assigned to each node v of G (except the root) arbitrarily choose one of the edges directed into v.

For each node v in G, there is a unique sequence of spanning tree edges from the root of G to v. We use $\rho(v)$ to denote the sequence of edge labels along this path. An *embedding* is a mapping d from nodes of G to nodes of H such that d maps the root of T to the root of H and if v and u satisfy $\rho(v) = \rho(u) \cdot i$ then there is an edge labeled i from $d(u)$ to $d(v)$. In other words, $d(u)$ is that node in H reached by following the edge sequence $\rho(u)$ from the root of H. If T cannot be embedded in H then there is a sequence of edge labels present in G but absent from H; hence G does not match H.

2. Embed T in H, if possible, and let d be the resulting embedding. Check that for all nodes u of T, u and $d(u)$ are labeled by the same function symbol. If there is no embedding or if the check fails, then output "no match."
3. For each node v of G labeled by a variable, say x, let $\sigma(x)$ be the term rooted at $d(v)$ in H. Apply the substitution σ to G by replacing all edges of the form (u, v) by $(u, d(v))$. Let $C = \sigma(G)$ denote the dag obtained from G by performing the substitution.
4. Test dags C and H for equivalence. G matches H if and only if $C \equiv H$.

Note that the mapping σ can be described by specifying, for each variable x labeling a node v of G, the pair $(x, d(v))$.

To prove correctness of the algorithm we need the following lemma.

LEMMA 11. *Let G, H, and C be as in Algorithm Match. Then G matches H if and only if $C \equiv H$.*

Proof. Clearly, if $C \equiv H$ then G matches H, as then $C \equiv \sigma(G) \equiv H$, where σ is the substitution described in step 3.

To prove the opposite direction we note that if G matches H, then the substitution σ found by the algorithm is the only one possible. Suppose otherwise that there was some other σ' such that $\sigma'(G) \equiv H$. Let x be any variable of G and let v be the node labeled by x. Since the roots of $\sigma'(G)$ and H are equivalent, the two nodes reached by following the path $\rho(v)$ from both roots must be equivalent. These nodes are v in G and $d(v)$ in H, where d is the embedding found by the algorithm. Therefore, $\sigma'(x) = \sigma(x)$. □

To prove Theorem 9 we need only prove we can perform the embedding (step 2) in $\mathrm{NC}^2(n^2)$. To do this, we will need a few lemmas.

LEMMA 12. *Given a directed path p with n nodes whose nodes are labeled with function symbols and whose edges are labeled with integers, and given a term dag H with k nodes ($k \geq n$), there is an algorithm which checks whether p can be embedded in H (and produces an embedding if there is one) in* $\mathrm{NC}^1(kn)$.

Proof. We form the product graph $Z = p \times H$ whose nodes are all pairs of nodes (u, v) where $u \in p$ and $v \in H$. There is an edge labeled i from (u, v) to (u', v') in Z if and only if there are edges (u, u') in p and (v, v') in H both labeled i. Let p_1 and p_n be the initial and final nodes of p, respectively, and let r be the root of H. Then p can be embedded in H if and only if there is a path in Z from (p_1, r) to (p_n, w) for some node w of H.

Note that since p is a path, each node in the product graph Z has outdegree at most 1. We can therefore apply the standard technique of "pointer chasing." We assign a processor to each node of Z. These processors initialize two arrays; a bit array E which indicates whether a node is known to be reachable from (p_1, r) (should one exist), and a successor array S.

1. Initialization:
   If $a = (p_1, r)$ then $E(a) := 1$, else $E(a) := 0$.
   If there is an edge of Z from a to b, then $S(a) := b$.
   For all nodes w in H, $S((p_n, w)) := (p_n, w)$.
2. Repeat $\lceil \log n \rceil$ times:
   If $E(a) = 1$ then $E(S(a)) := 1$;
   $S(a) := S(S(a))$.
3. Test for embedding:
   If $S((p_1, r)) = 0$ then output "No embedding" and halt.
4. Find embedding given that one exists:
   If $E((u, v)) = 1$ then v is the embedding of u.

The number of nodes a for which $E(a) = 1$ doubles at each step until all nodes reachable from (p_1, r) are found. Thus, after execution of the algorithm $E(a) = 1$ if and only if a is reachable from (p_1, r). Moreover, after execution of the algorithm $S((p_1, r)) \neq 0$ if and only if a node of the form (p_n, w) is reachable from (p_1, r). □

LEMMA 13 (Tarjan and Vishkin [16]). *Let T be a tree of size m. There are* $\mathrm{NC}^1(m)$ *algorithms which produce* (a) *the depth-first numbering of T and,* (b) *for each node u of T, the number of nodes in the subtree rooted at u.*

The following lemma shows that the embedding of a tree in a dag can be accomplished in $\mathrm{NC}^2(n^2)$, completing the proof of Theorem 9.

LEMMA 14. *The problem of embedding a tree T having m nodes in a dag H having k nodes is in* $\mathrm{NC}^2(km)$.

Proof. The embedding is done by a recursive divide-and-conquer approach.

1. For each node $v \in T$ compute the size of the subtree rooted at v. This can be done in $\mathrm{NC}^1(m)$ by Lemma 13(b). Let the *weight* of a directed edge (u, v) be the size of the subtree rooted at v. Consider any path originating at the root of T defined by selecting the heaviest edge out of each node to be in the path (ties are broken arbitrarily). Let us call such a path a *heavy path.*
2. Embed the heavy path. This can be done in $\mathrm{NC}^1(km)$, by Lemma 12.
3. For each node v in the heavy path embedded in step 2, embed all the children of v. This can be accomplished in $\mathrm{NC}^0(km)$.
4. For each node u embedded in step 3, embed the subtree rooted at u in the subdag rooted at $d(u)$ where d is the embedding constructed in step 3. These embeddings are performed recursively in parallel.

Although the number of nodes not embedded by the end of step 3 may be quite large, each remaining *subtree* contains at most $m/2$ nodes. Letting $\mathrm{Time}(m)$ be the parallel time to embed a tree of size m, this yields the recurrence relation

$$\mathrm{Time}(m) = c \log m + \mathrm{Time}(m/2),$$

which has solution $O(\log^2 m)$. Since the subtrees embedded in step 4 are pairwise disjoint, it is easy to see that km processors suffice to do all these embeddings in parallel. □

3.2. Special cases of term matching. In this section, we describe algorithms for checking equivalence of constant dags C and H (as obtained in step 3 of Algorithm Match) when H is of a special form. When used in step 4 of Algorithm Match these yield improved results for term matching. We assume as in § 3.1 that the two input dags are disjoint.

Let u and v be nodes of a term dag. We say $u \equiv v$ if the term represented by the subdag rooted at u equals the term represented by the subdag rooted at v. A dag is *compact* if $u \equiv v \Rightarrow u = v$, for all nodes u and v.

The proof of the following lemma is straightforward.

LEMMA 15. *Let G be an arbitrary dag with spanning tree T. Let H be a second dag and let d be an embedding of T in H. Then G matches H if and only if the following two conditions are satisfied.*

(a) *For each node v of G not labeled with a variable name, the label of v equals the label of $d(v)$.*

(b) *For each edge $e = (u, v)$ in G but not in the spanning tree T, $d(u)_i \equiv d(v)$, where i is the label of e and $d(u)_i$ denotes the ith child of $d(u)$ in H.* □

Lemma 15 implies term matching is particularly easy when H is a compact dag, for in that case checking conditions of the form $d(u)_i \equiv d(v)$ reduces to checking $d(u)_i = d(v)$.

COROLLARY 16. *The problem of determining whether an arbitrary dag matches a compact dag is in* $\mathrm{NC}^2(n^2)$. □

A slightly harder case is when H is a tree. However, since the tree representation of a term is unique, testing equivalence of trees (as required by Condition (b) of Lemma 15) reduces to checking that the trees are identical, and while this is more difficult than checking equality of nodes we can solve it efficiently using some of the results of Tarjan and Vishkin stated in Lemma 13. In effect, we will change H into a compact dag by determining, for all nodes u and v of H, whether $u \equiv v$.

The following uses the depth-first numbering produced by Lemma 13(a). This numbering respects the edge labels, in the sense that for each node v the numbers in the subtree rooted at the ith child of v are smaller than those assigned to nodes in the subtree rooted at the $(i+1)$st child.

LEMMA 17. *Checking equivalence of trees of size n is in* $\mathrm{NC}^1(n)$.

Proof. We will show that the depth-first numbers and node labels of the nodes in a tree completely specify the tree. The lemma then follows by Lemma 13(a).

Given a tree T of size n we show by induction on k that the subtree consisting of the first k nodes in the depth-first numbering of T can be recreated from the depth-first numbers and labels of these nodes. In the following, let "node i" denote the node in T with depth-first number i.

The basis $k=1$ is trivial, since the root is always labeled 1.

$k>1$. We assume the result for subtrees of size $k-1$ and prove it for k. Given the subtree S consisting of the first $k-1$ nodes of T, we find the largest $i \leqq k$ such that m, the number of children of node i in S, is less than the arity of the label of node i. Then node k is the $(m+1)$st child of node i. Clearly, k cannot be the child of any node with depth-first number greater than i, since all of these have all their children in S. On the other hand, node k cannot be the child of any node with depth-first number less than i. To see this, assume for the sake of contradiction that node k is the child of a node u, where $u<i$. Let w be the least common ancestor of u and i. Let j be the label of the edge from w on the path to i. By the inductive hypothesis, w was correctly given its first j children in the construction of S. In particular, the jth child of w is added only if the subtrees rooted at the first $j-1$ children are complete. Thus, if u does not have all its children in S it must be that $u=w$. In this case, the first available slot for another child of u is the $(j+1)$st. But all descendants of the jth child of u must have numbers smaller than that of the $(j+1)$st, so if i does not have all its children in S, k cannot be a child of u. □

Our approach to matching a general dag with a tree is to find all equivalent nodes, thereby effectively turning the tree into a compact dag. We first eliminate many pairs of nodes from consideration since they are obviously not equivalent, and then apply Lemma 17 in parallel to the remaining pairs.

The *size* of a node v, denoted $\mathrm{size}(v)$, is the number of nodes in the subtree rooted at v. Clearly, nodes u and v can be equivalent only if $\mathrm{size}(u)=\mathrm{size}(v)$, but the converse is false. However, this observation allows us to bound the total cost of checking the subtree equivalences to yield the following.

LEMMA 18. *The problem of determining all equivalent nodes of a tree is in* $\mathrm{NC}^1(n^2)$, *where n is the size of the tree.*

Proof. The general idea is to apply Lemma 17 in parallel to all pairs of nodes u, v for which $\mathrm{size}(u)=\mathrm{size}(v)$. However, we must be careful, as a brute force analysis leads to a processor bound of $O(n^3)$. For each i, $1 \leqq i \leqq n$, let n_i denote the number of nodes of size i. By Lemma 17, the number of processors sufficient to check equivalence of all pairs u, v of nodes of size i is in_i^2. Thus, the total number of processors needed to find all equivalent nodes is $\sum_i in_i^2$.

Now, $\sum_i n_i \leqq n$, and for each i, $in_i \leqq n$. Multiplying each side of the last inequality by n_i yields $in_i^2 \leqq nn_i$. Putting all this together we obtain

$$\sum_i in_i^2 \leqq \sum_i nn_i \leqq n \sum_i n_i \leqq n^2.$$ □

Applying the algorithm of Lemma 18 and then using the approach of Lemma 15 proves that testing whether an arbitrary dag matches a tree is in $\mathrm{NC}^2(n^2)$. In fact we can do better in terms of time.

THEOREM 19. *Term matching for a general dag G and a constant tree H can be done in* $\mathrm{NC}^1(n^2)$.

Proof. We only have to argue for $O(\log n)$ instead of $O(\log^2 n)$ parallel time.

In the case that H is a tree, the only part of the algorithm which uses time $\log^2 n$ is the embedding of the spanning tree T in the tree H. Since T and H are both trees, the embedding can be done in time $O(\log n)$ as follows.

Construct the product graph Z whose nodes are all pairs (u, v) such that u is a node of T and v is a node of H. There is an edge labeled i directed from (u, v) to (u', v') if there is an edge labeled i from u to u' in T and an edge labeled i from v to v' in H. Since T and H are both rooted trees, it is easy to see that Z is a forest of rooted trees. Letting r_1 and r_2 be the roots of T and H, respectively, we want to find all nodes of Z which are reachable from (r_1, r_2) since the embedding maps u to v if and only if (u, v) is reachable from (r_1, r_2). Since Z contains at most $m = n^2$ nodes this can be done in $\mathrm{NC}^1(n^2)$ as follows.

We first transform Z from a forest of trees to a single tree by creating a new root and creating an edge from the new root to each of the roots in Z. For simplicity, let Z denote the resulting tree. We then compute a depth-first numbering of Z and size(a) for each node a in Z. By Lemma 13 these tasks can be accomplished in $\mathrm{NC}^1(m)$. Let a be a node of Z and let k be the depth-first number of a. Then the descendants of a are those nodes with depth-first numbers $k, \cdots, k + \mathrm{size}(a) - 1$. □

Verma, Krishnaprasad, and Ramakrishnan [17] have recently shown that term matching of two trees is in $\mathrm{NC}^2(n)$ (whereas $\mathrm{NC}^1(n^2)$ is a corollary of Theorem 19).

3.3. Another special case of unification in NC. In this section, we examine unification of two terms in the special case that the terms share no variables and at least one of the terms is linear. In performing unification on term dags we need a way of representing the result. In general, two terms are unifiable if and only if a certain type of equivalence relation can be constructed on the nodes of the labeled dag representing these terms. Given this relation, we can define the *reduced* graph, obtained by coalescing all equivalent nodes into a single node. We can extract a unifier σ from the reduced graph by taking $\sigma(x)$ to be the term in the reduced graph that is represented by the node formed from the equivalence class of x.

A relation R on the nodes of a term dag is a *correspondence* relation if for all pairs of nodes u, v in the dag

$$uRv \Rightarrow u_i R v_i,$$

where u_i and v_i are corresponding children, respectively, of u and v. A correspondence relation that is also an equivalence relation will be called a *c-e relation.*

A relation R on the nodes of a term dag is *homogeneous* if for all pairs of nodes u, v which are labeled by function symbols we have

$$uRv \Rightarrow \mathrm{label}(u) = \mathrm{label}(v).$$

An equivalence relation R on the nodes of a dag is *acyclic* if the R-equivalence classes are partially ordered by the arcs of the dag. Paterson and Wegman [12] have shown that if u and v are nodes of a labeled dag G then the terms represented by the subdags with roots u and v, respectively, are unifiable if and only if there exists an acyclic homogeneous c-e relation R on the nodes of the dags satisfying uRv. Since we are considering the special case where one of the terms is linear and the two terms do not have any variables in common, it is easy to see that cyclic c-e relations cannot arise, so we do not mention the acyclicity condition further.

A substitution σ is *more general* than a substitution τ if there exists a substitution ρ with $\tau = \rho \circ \sigma$. If R is the minimal c-e relation with uRv then the unifying substitution obtained from the reduced graph defined by R as described above is the most general unifier. The *size* of the most general unifier is the number of nodes in the reduced graph obtained by coalescing nodes equivalent under R, where R is minimal.

LEMMA 20. *Unification of two linear terms represented by trees with no shared variables can be solved in* $\mathrm{NC}^1(n^2)$. *Moreover, the most general unifier is linear and has size at most the sum of the sizes of the original trees.*

Proof. Let T_1 and T_2 be trees representing linear terms with no shared variables, and n be the size of the instance. As in the proof of Theorem 19, we construct the product graph Z whose nodes are all pairs of the form (u, v), where u is a node of T_1 and v is a node of T_2. There is an edge labeled i directed from (u, v) to (u', v') in Z if there are edges labeled i from u to u' in T_1 and from v to v' in T_2. As in the proof of Theorem 19, letting r_1 and r_2 denote the roots of the two trees, respectively, we find all nodes in the product graph reachable from (r_1, r_2). We do not repeat the details here. Let R be the relation on nodes of $T_1 \cup T_2$ defined by: uRv if and only if (u, v) is reachable in Z from (r_1, r_2). For all pairs of internal nodes u, v we see that $uRv \Rightarrow u_iRv_i$, where u_i and v_i are corresponding children of u and v, respectively. It is also clear that because each node in the union of the original trees is related to at most one other node, R is an equivalence relation (trivially). We note that R is the minimal c-e relation in which the two roots are related. Once we have constructed Z and determined R, we can easily check R for homogeneity. If so, then as shown in [12], the two trees are unifiable. We construct the reduced graph and, from it, the most general unifier, as described in the beginning of this section.

Because Z contains at most n^2 nodes, this can all be done in $\mathrm{NC}^1(n^2)$. Further, because the reduced graph contains no more nodes than the union of T_1 and T_2, the most general unifier has size at most the sum of the sizes of the two trees. Finally, the most general unifier is linear since it is obtained by a substitution which maps each variable to a linear term such that different variables are mapped to linear terms involving disjoint sets of variables. □

Using Lemma 20 we can now prove the main result of this section.

THEOREM 21. *Let T be a tree representing a linear term and H be an arbitrary rooted dag sharing no variables with T. Then the unification problem for T and H can be solved in* $\mathrm{NC}^2(n^2)$.

Sketch of Proof. Without loss of generality we assume that for each variable x there is at most one node of H labeled with x. Let r_1 and r_2 be the roots of T and H, respectively. Again we are searching for the minimal c-e relation R on the nodes of $T \cup H$ such that r_1Rr_2. We first modify the recursive embedding technique of Lemma 14 to handle the embedding of a tree in a dag when the dag is not necessarily constant, as in the present case. Thus, there could be some nodes of T that cannot be embedded in H because some path in H ends with a node labeled by a variable while the corresponding path in T continues. As in the case of Lemma 14, the modified algorithm is in $\mathrm{NC}^2(n^2)$. We define R to be the reflexive transitive closure of R', where $uR'v$ if u is mapped to v by the embedding. Check R for homogeneity.

If uRv where u is a node of T and v is a node of H, and u is labeled by a variable x, then we define $\sigma(x)$ to be the term represented by the subdag rooted at v. Because there is a unique path to a node labeled x in T there is nothing to check. More interesting is the case when there exist several nodes $u_1, \cdots, u_k$ in T and a node v in H labeled with a variable y such that u_iRv, $i = 1, \cdots, k$. This can happen if there are k paths to v in H. In this case we must unify all k trees rooted at the u_i. However,

because T is linear these subtrees share no variables, so we can apply Lemma 20. In order to perform all the unifications quickly in parallel, we split the subtrees into at most $k/2$ pairs, and unify all pairs in parallel. By Lemma 20 the size of the most general unifier for each pair is no larger than the sum of the sizes of the original trees. We can therefore apply the lemma recursively on the $\leqq \lceil k/2 \rceil$ remaining trees. Because $k \leqq n$ and each application of Lemma 20 can be performed in $\mathrm{NC}^1(n^2)$ the entire procedure is in $\mathrm{NC}^2(n^2)$. □

Remark. By modifications to the proofs of Lemma 20 and Theorem 21, it is easy to see that unification of a linear term with an arbitrary term can be done in NC even if both terms are represented by general dags. An outline of the algorithm follows. Let G and H be the given dags with roots r_1 and r_2, respectively, where G represents a linear term and H represents an arbitrary term. Form the product graph Z as in the proof of Lemma 20, and solve a reachability problem (in $\mathrm{NC}^2(M(n^2))$) to find all nodes of Z reachable from (r_1, r_2). Let R be the reflexive transitive closure of the relation R' defined by $uR'v$ if and only if (u, v) is reachable from (r_1, r_2). Proceeding as in the proof of Theorem 21, the only difference is the case where there are several nodes $u_1, \cdots, u_k$ in G and a node v in H labeled with a variable such that $u_i R v$ for all i. Now the u_i are roots of subdags which represent linear terms; as before, these subdags share no variables. By again solving a reachability problem on a product graph, the unification problem for two linear terms which do not share variables and which are represented by general dags can be solved in NC. As before, the most general unifier is linear and its size is at most the size of the union of the two dags (nodes which appear in both dags are counted only once in the union).

4. Unification of linear terms is complete for P. Recall that a term is linear if no variable appears more than once in the term.

THEOREM 22. *Unification is* P*-complete even if both terms are linear, are represented by trees, and have all function symbols with arity* $\leqq 2$.

Proof. The proof is by a reduction from the circuit value problem (CVP) which was proved P-complete by Ladner [8]. Because of the nature of our reduction, it is useful to require that instances of CVP be in a particular form described next. An instance of CVP is a dag whose nodes are of four types. An *input node* has no edges directed in and one edge directed out; this edge is called an *input edge.* An *output node* has one edge directed in and no edges directed out; each dag has exactly one output node and the edge directed into this node is called the *output edge.* A NAND *node* has two edges directed in and one edge directed out. A *fan-out node* has one edge directed in and any nonzero number of edges directed out. In addition, each input edge is labeled with a Boolean value, either 0 (false) or 1 (true). Given the assignments of Boolean values to the input edges, Boolean values are associated with all the other edges in the obvious way: the value of the edge directed out of a NAND-node is the Boolean NAND of the values of the two edges directed in; the value of all edges directed out of a fan-out node is the same as the value of the edge directed in. The problem CVP is to recognize the set of instances such that the output edge has value 1. (Although Ladner's proof of the P-completeness of CVP uses Boolean functions other than NAND, any such function can be built from a small number of NANDs and Boolean constants, so CVP as defined above is P-complete.)

Given an instance G of CVP, we transform it to a pair of linear tree terms, T_1 and T_2, with roots r_1 and r_2, respectively. For simplicity, we let the two trees have function symbols with arities greater than 2. The trees can then be further transformed by replacing each subterm $f^{(k)}(t_1, t_2, \cdots, t_k)$ involving a k-ary function symbol by

the term $f(t_1, f(t_2, \cdots, f(t_{k-1}, t_k)) \cdots)$ involving the 2-ary function symbol f. In addition to the two roots, T_1 has nodes A_e and B_e and T_2 has nodes C_e and D_e for each edge e of G. Unifications among these four nodes encode the Boolean value of e as follows (in this proof we indicate a unification between two nodes by writing $\sim$ between them): if e has value 0, then $A_e \sim D_e$ and $B_e \sim C_e$; if e has value 1, then $A_e \sim C_e$ and $B_e \sim D_e$. The appropriate unifications of nodes corresponding to input edges are forced by making these nodes be corresponding children of the roots. For example, if e is the ith input edge and if e is assigned value 1, then there is an edge labeled $2i-1$ from r_1 to A_e, an edge labeled $2i-1$ from r_2 to C_e, an edge labeled $2i$ from r_1 to B_e, and an edge labeled $2i$ from r_2 to D_e.

For each fan-out node and NAND node of G, edges and nodes are added to the trees to force the unifications encoding Boolean values to be propagated correctly. For each fan-out node of G, if the node has edge e directed in and edges $e_1, \cdots, e_k$ directed out, then for each i with $1 \leqq i \leqq k$, there is an edge labeled i from A_e to A_{e_i}, from B_e to B_{e_i}, from C_e to C_{e_i}, and from D_e to D_{e_i}. For each NAND node, with edges e' and e'' directed in and edge e directed out, the trees contain the nodes and edges shown in Fig. 3 (to simplify notation, A' is written for $A_{e'}$ etc.).

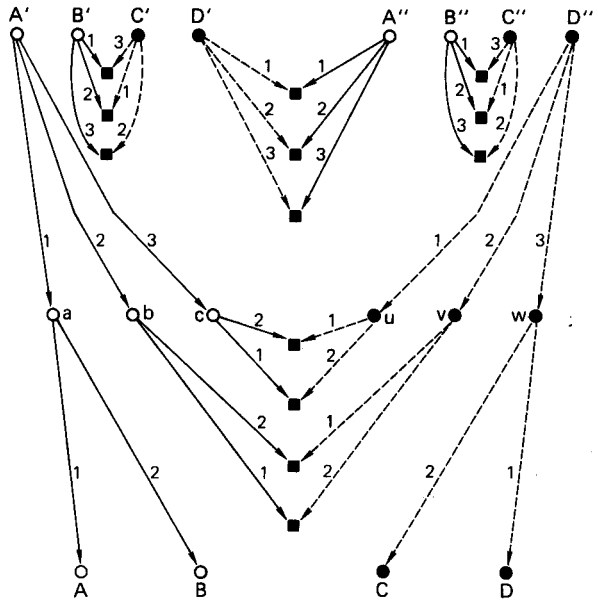

FIG. 3. *The transformation of a* NAND *node used in the proof of Theorem* 22. *Nodes of* T_1 (*resp.*, T_2) *are drawn as open circles* (*resp., solid circles*). *Edges of* T_1 (*resp.*, T_2) *are drawn as solid lines* (*resp., dashed lines*). *Common leaves which are labeled by variables are drawn as solid squares.*

We must also define the labeling of tree nodes. Each nonleaf node is labeled by the function symbol $f^{(k)}$ where k is the outdegree of the node. The leaves which are drawn as squares in Fig. 3 are each labeled by a different variable symbol. If p is the output edge of G, then A_p and C_p are labeled with the constant symbol g, and B_p and D_p are labeled with a different constant symbol h. This completes the description of the transformation.

We must argue that the output edge of G has value 1 if and only if T_1 and T_2 are unifiable. To do this it is sufficient to show that the transformations of fan-out nodes and NAND nodes correctly propagate Boolean values according to the encoding

of values by unifications described above. For fan-out nodes this is obvious. To verify this for the NAND construction in Fig. 3, it is helpful to break the construction into two parts. The first part, which consists of the part of the figure above the nodes marked a, b, c, u, v, w (including these six nodes), computes the numerical sum of the two input values, viewing these values as integers rather than Boolean values. The three possible sums, 0, 1, and 2, are encoded by unifications among a, b, c, u, v, w. It is easy to check that the four possible values for e' and e'' force unifications among a, b, c, u, v, w as follows:

$$e'=0 \text{ and } e''=0 \quad \text{implies} \quad a\sim u,\ b\sim v,\ c\sim w,$$

$$e'=0 \text{ and } e''=1 \quad \text{implies} \quad a\sim v,\ b\sim w,\ c\sim u,$$

$$e'=1 \text{ and } e''=0 \quad \text{implies} \quad a\sim v,\ b\sim w,\ c\sim u,$$

$$e'=1 \text{ and } e''=1 \quad \text{implies} \quad a\sim w,\ b\sim u,\ c\sim v.$$

We can now forget about the top half of Fig. 3 and just check that these three possible unifications among a, b, c, u, v, w force the proper unifications among A, B, C, D. The verification of the following is again straightforward:

$$a\sim u,\ b\sim v,\ c\sim w \quad \text{implies} \quad A\sim C \text{ and } B\sim D\ (e=1),$$

$$a\sim v,\ b\sim w,\ c\sim u \quad \text{implies} \quad A\sim C \text{ and } B\sim D\ (e=1),$$

$$a\sim w,\ b\sim u,\ c\sim v \quad \text{implies} \quad A\sim D \text{ and } B\sim C\ (e=0).$$

If the output edge p of G has value 0, then an attempt to unify the roots r_1 and r_2 will force the unification of two nodes A_p and D_p labeled with different constant symbols. On the other hand, if the output edge has value 1, then the two roots can be unified. □

In the case that the two terms do not share any variables, we have noted in § 3.3 that unification can be solved in NC if one of the terms is linear. The following easy corollary of Theorem 22 shows that this is in some sense the best possible, since if both terms are barely nonlinear the problem becomes P-complete.

THEOREM 23. *Unification is P-complete even if both terms are represented by trees, no variable appears in both terms, each variable appears at most twice in some term and all function symbols have arity* $\leqq 2$.

Proof. As in the previous proof, we allow function symbols with large arities. The proof is by a reduction from the unification problem for linear trees. Let T_1 and T_2 be a given pair of trees representing linear terms, and let $x_1, x_2, \cdots, x_m$ be the variables which appear in both trees. For each i with $1 \leqq i \leqq m$, replace the single occurrence of x_i in T_1 by a new variable x_{i1} and replace the single occurrence of x_i in T_2 by x_{i2}. For each i, we can force x_{i1} and x_{i2} to be equal by increasing the arity of the roots from 2 to $2+m$, adding a new edge labeled $i+2$ from the root of T_1 to a new node labeled x_{i1}, and adding a new edge labeled $i+2$ from the root of T_2 to a new node labeled x_{i2}. Clearly, the transformed trees are unifiable if and only if T_1 and T_2 are unifiable. □

REFERENCES

[1] S. A. AMITSUR AND J. LEVITZKI, *Minimal identities for algebras*, Proc. Amer. Math. Soc., 1 (1950), pp. 449-463.

[2] W. F. CLOCKSIN AND C. S. MELLISH, *Programming in Prolog*, Springer-Verlag, New York, Berlin, 1981.

[3] D. COPPERSMITH AND S. WINOGRAD, *On the asymptotic complexity of matrix multiplication*, SIAM J. Comput., 11 (1982), pp. 472-492.

[4] C. DWORK, P. C. KANELLAKIS, AND J. C. MITCHELL, *On the sequential nature of unification*, J. Logic Programming, 1 (1984), pp. 35–50.

[5] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, Proc. 10th ACM Symposium on Theory of Computing, 1978, pp. 114–118.

[6] G. HUET AND D. OPPEN, *Equations and rewrite rules: a survey*, in Formal Language Theory: Perspectives and Open Problems, R. V. Book, ed., Academic Press, New York, 1980.

[7] R. KOWALSKI, *Predicate logic as a programming language*, Proc. IFIPS, 74 (1974), pp. 569–574.

[8] R. LADNER, *The circuit value problem is log space complete for* P, SIGACT News, 7 (1975), pp. 18–20.

[9] J. MALUSZYNSKI AND H. J. KOMOROWSKI, *Unification-free execution of horn-clause programs*, Proc. 2nd IEEE Logic Programming Symposium, 1985, pp. 78–86.

[10] R. MILNER, *A theory of type polymorphism in programming*, J. Comput. System Sci., 17 (1978), pp. 348–375.

[11] V. PAN AND J. REIF, *Efficient parallel solution of linear systems*, Proc. 17th Annual ACM Symposium on Theory of Computing, 1985, pp. 143–152.

[12] M. S. PATERSON AND M. N. WEGMAN, *Linear unification*, J. Comput. System Sci., 16 (1978), pp. 158–167.

[13] M. O. RABIN, *Probabilistic algorithm for testing primality*, J. Number Theory, 12 (1980), pp. 128–138.

[14] J. A. ROBINSON, *A machine oriented logic based on the resolution principle*, J. Assoc. Comput. Mach., 12 (1965), pp. 23–41.

[15] J. T. SCHWARTZ, *Fast probabilistic algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27 (1980), pp. 701–717.

[16] R. E. TARJAN AND U. VISHKIN, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput., 14 (1985), pp. 862–874.

[17] R. M. VERMA, T. KRISHNAPRASAD, AND I. V. RAMAKRISHNAN, *An efficient parallel algorithm for term matching*, Dept. of Computer Science, State University of New York, Stony Brook, New York, 1986, manuscript.

[18] H. YASUURA, *On the parallel computational complexity of unification*, Res. Report ER 83-01, Yajima Laboratories, October 1983.