May 1987

Report No. STAN-CS-87-f156

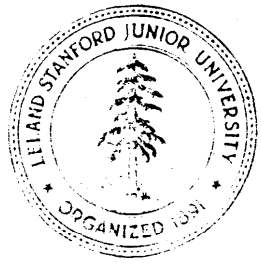
The Dynamic 'Tree Expression Problem

by

Ernst W. Mayr

Department of Computer Science

Stanford University
Stanford, CA 94305



The Dynamic Tree Expression Problem

Ernst W. Mayr
Stanford University

May 2, 1987

Abstract

We present a uniform method for obtaining efficient parallel algorithms for **a** rather large class of problems. The method is based on a logic programming model, and it derives its efficiency from fast parallel routines for the evaluation of expression trees.

This work was supported in part by a grant from the AT&T Foundation, ONR contract N00014-85-C-0731, and NSF grant DCR-8351757.

1 Introduction

Computational complexity theory tries to classify problems according to the amount of computational resources needed for their solution. In this sense, the goal of complexity theory is a refinement of the basic distinction between decidable (solvable, **recursive)** and undecidable (unsolvable, non-recursive) problems. There are basically three types of results: (i) algorithms or upper bounds showing that a certain problem can indeed be solved using a certain amount of resources; (ii) lower bounds establishing that **some** specified number of resources is insufficient for the problem to be solved; and (iii) reductions of one problem to another, showing that the first problem is essentially no harder than the second. Complexity classes are collections of all those problems solvable within a specific bound, and complete problems in a complexity class are distinguished problems such that every other problem in the class is efficiently reducible to them. In this sense, they are hardest problems in the class.

A very important goal of computational complexity theory is to characterize which problems are *feasible,* i.e. solvable efficiently in an intuitive sense. For (standard) sequential computation, the class $\mathcal{P}$ of those problems solvable in polynomial time has become accepted as the "right" class to represent feasibility. For parallel computation, on a wide number of machine models (idealized and realistic), the class $\mathcal{NC}$ is considered to play the same role. It is the class of problems solvable (in the PRAM model, discussed below) in time polylogarithmic in the size of the input while using a number of processing elements polynomial in the input size.

In this paper, we show an interesting relationship between efficient parallel computation and the existence of reasonably sized *proof trees.* While the reader may feel reminded of the well-known characterization of the class $\mathcal{NP}$ as those problems permitting short (polynomial length) proofs of solutions, proof trees are considerably more restricted in the sense that if some intermediate fact is reused the size of its (sub) proof tree is counted with the appropriate multiplicity. Drawing on, and generalizing, work reported in [14], [18], and [21], we give a very general scheme to obtain efficient parallel algorithms for many different problems in a way that abstracts from details of the underlying parallel architecture and the necessary synchronization and scheduling.

The remainder of the paper is organized as follows. In the next section, we discuss a few techniques for parallel algorithm design appearing in the literature. Then we introduce our new technique, the *dynamic tree expression problem* or *DTEP,* and present some of its' applications and extensions. In particular, we show an adaptation of the basic DTEP paradigm to the evaluation of straightline programs computing polynomials. The final section summarizes our results and mentions some open problems.

2 Techniques for Parallel Algorithm Design

General techniques for the design of efficient parallel algorithms are hard to come by. To a large degree, this is due to the fact that there are so many vastly different parallel machine architectures around, and no compilers between the programmers and the machines that

could provide a uniform interface to the former. Obviously, this is a situation quite different from sequential computation. We may even ask ourselves why we should be looking for such a compiler, or whether parallel algorithms shouldn't be developped for specific architectures in the first place. Even more so, we might ask whether parallelism is any good for solving problems in general, or whether there are just a few selected, and highly tuned, applications that could benefit from heavy parallelism. We believe that there is a large number of problems and applications out there which should be parallelized individually, designing architectures permitting the highest speedup for these applications. There is also a large class of applications whose data structures and flow of control are extremely regular and can easily be mapped onto appropriate parallel machine architectures, like systolic arrays or n-dimensional hypercubes. It is our impression, however, that many other problems, particularly in Artificial Intelligence, exhibit the same specifications (with respect to parallelization) as we would expect from general purpose parallel computation. It is the interest in these, and other, applications that makes us convinced that general purpose parallelism is relevant, and that in fact it carries immense potential for areas of general importance.

The design of parallel algorithms may be guided by several, sometimes conflicting, goals. The first, and maybe most obvious, goal in using parallel machines may be to obtain fast, or even the fastest possible, response times. Such an objective will be particularly important in security sensitive applications, like warning or monitoring systems; weather forecasters also seem to be interested in this particular aspect of parallel computation. Another option seems to worry about the marginal cost factor: how much real computational power do we gain by adding one (or k) additional processors to the system? While optimal speedup can be achieved over a certain range of execution times (depending on the problem), fastest execution times may be obtained, for certain problems, only by using a very large number of parallel processors, thus decreasing, in effect, the actual speed-up.

Another consideration is what could be termed the *Law of Small Numbers.* Many of the very fast parallel algorithms have running times which are some power of the logarithm of the input size n, like $O(\log^2 n)$. If we compare this running time with one given by, say, $O(\sqrt{n})$, and assume that the constant factor is 1 in both cases for simplicity, we get that the *polylogarithmic* algorithm is actually slower than the polynomial algorithm for n greater than 2 and less than 65,536.

2.1 Parallel Machine Models

We shall base most of our discussions onto a theoretical machine model for parallel computation called the *Parallel Random Access Machine,* or *PRAM* [4]. This machine model has the following features. There is an unbounded number of identical *Random Access Machines* (or *RAM's),* and an unbounded number of global memory cells, each capable of holding an arbitrarily big integer. The processors work synchronously, controlled by a global clock. Different processors can execute different instructions. Each processor can access (read or write) any memory cell in one step. Our model also allows that more than one processor read the same memory cell in one step (concurrent *read),* but

it disallows *concurrent writes* to the same memory cell. For definiteness, we assume that all reads of global memory cells in a step take place before all writes (resolving *read/write conflicts).*

Most of our algorithms actually do not use concurrent reads, and can be run on a somewhat weaker machine model in which concurrent reads are forbidden. In any case, algorithms in our model can always be simulated on this weaker model with a very small slowdown.

The shared memory feature of the PRAM model, allowing unbounded access to memory in a single step, is somewhat idealistic. A more realistic machine model consists of a *network* of (identical) processors with memory modules attached to them. The processors are connected via point-to-point communication channels. Each processor can directly access only cells in its own memory module, and it has to send messages to other processors in order to access data in their modules. To respect technological constraints, the number of channels per processor is usually bounded or a very slowly growing function of the number of processors. Examples for such networks of processors are the Hypercube [20], the Cube-Connected-Cycles network [17], or the Ultracomputer (RP3) [15], [19].

2.2 Parallel Complexity Measures

The two most important parameters determining the computational complexity of algorithms in the PRAM model are the time requirements and the number of processors used by the algorithm. Both are usually expressed as functions of the input size n. Note that another factor determining the practical efficiency of a parallel algorithm — its communication overhead — is of no importance in the PRAM model. As in sequential computation, and its associated complexity theory, complexity classes are defined based on bounds of the available resources, like time or number of processors. One such class has become particularly important in the theoretical study of parallel algorithms and the inherent limits of parallelism. This class is called *NC* and is defined as the class of problems solvable on a PRAM in time ***polylogarithmic*** (polylog for short) in the size of the input while using a number of processors polynomial in the input size. Thus, a problem is in *NC* if there is a PRAM algorithm and some constant $c > 0$ such that the algorithm runs in time $O(\log^c n)$ and uses $O(n^c)$ processors on instances of size n.

The resource constraints for *NC* reflect two goals to be achieved by parallelism: a (poly)logarithmic running time represents an "exponential" speedup over sequential computation (requiring at least linear time); and the restriction on the number of processors limits the required hardware to be "reasonable". Of course, both these considerations have to be taken in the asymptotic sense, and are thus of a rather theoretical nature (see the Law of Small Numbers above).

However, the class *NC* is *robust* in the sense that it does not change with small modifications in the underlying machine model. In fact, it is the same for the PRAM model(s) outlined above as well as the Hypercube or Ultracomputer architecture. It was originally defined for boolean circuits in [16]. Membership in *NC* has become (almost) synonymous with "efficiently parallelizable" .

Real parallel machines have, of course, a fixed number of processors. Even though some parallel algorithm may have been designed for a (theoretical) parallel machine with a number of processors depending on the input size, the following theorem states that it is always possible to reduce, without undue penalty, the number of processors:

Simulation Theorem: An algorithm running in time T(n) on a p-processor PRAM, can be simulated on a $-processor PRAM, $p' \leq p$, in time $0(\lceil p/p' \rceil T(n))$. *cl*

If an algorithm runs in time T(n) on a sequential architecture, and we port it to a parallel architecture with p processors we cannot expect a parallel running time better than $\Omega(T(n)/p)$ — unless our machine models exhibit some strange quirks which are of no concern here. Conversely, we say that a problem of sequential time complexity $T(n)$ exhibits *optimal speedup* if its parallel time complexity on a PRAM using p processors is in fact $O(T(n)/p)$, for p in an appropriate range.

2.3 Parallel Programming Techniques

Only relatively few parallel programming schemes and / or techniques can be found in the literature. This is due in part to the fact that there is universal parallel architecture, and no commonly accepted way of hiding architectural details from parallel programs. Nonetheless, the PRAM model can, to a certain degree, play this role. Certain operations have been identified to be fundamental or important for **a** large number of parallel applications, and efficient algorithms have been developed for them. Examples are the parallel prefix computation [10], census functions [1 1], (funnelled) pipelines [7], parallel tree evaluation [13], [14], and sorting [2]. As an illustration, we state a procedural implementation of an algorithm for the parallel prefix problem, in a PASCAL like notation. Given an array $s_1, \ldots, s_n$ of elements from some arbitrary domain, and a binary operator o over that domain, this procedure computes all partial "sums" s_1 o . . . s_i, for $i = 1, \ldots, n$.

```
procedure census-function(n, s, result, o);
int n; gmemptr s, result; binop o;
co n is the number of elements in the input array starting at position s in global memory; result
is the index of the global memory cell receiving the result; o is an associative binary operator oc
begin
  local type_of_S: save, int: mask, myindex
  if PID < n then
    mask := 1; myindex := s + PID; save := M_myindex;
    while mask < n do
      if (PID & (2 * mask - 1) = 0) and PID + mask < n then
        M_myindex := M_myindex o M_myindex+mask
      fi;
      mask := 2 * mask
    od;
    if PID = 0 then M_result := M_s A;
```

if *myindex* $\neq$ result **then** $M_{myindex}$:= save fi
fi;
return
end *census-function.*

This algorithm takes, on a PRAM, O(logn) steps and uses *n* processors. As such, its speedup is suboptimal since the straightforward sequential algorithm runs in linear time. However, by grouping the input elements into contiguous groups of size $\lceil \log n \rceil$ (the last group may be smaller), we can solve the parallel prefix problem using just $\lceil n/\log n \rceil$ processors. Using one processor for every group, we first compute the prefix "sums" within every group. Then we run the above algorithm with the group prefix "sums" as input, and finally we extend the partial sums over the groups inside the groups, again using one processor per group. The time for this modified algorithm is still O(log n) (with a somewhat larger constant).

The parallel prefix problem has many applications. It occurs as a subproblem in algorithms for *ranking, linearization of lists, graph traversals, processor scheduling,* and others.

In the next section, we discuss another fundamental algorithm for parallel computation.

3 The Dynamic Tree Expression Problem

In [18], Alternating Turing machines using logarithmic space and a polynomial size computation tree are studied. These machines can be thought of as solving a recognition problem by guessing a proof tree and recursively verifying it. Each internal node in the proof tree is replaced, in a universal step, by its children in the tree while leave nodes correspond to axioms which can be verified directly. In every step, the machine is allowed to use only logarithmic storage (at every node) to record the intermediate derivation step.

We in effect turn the top-down computation of ATM's as discussed in [18] around into computations proceeding basically bottom-up, similar to the approach taken in [13], [14], and [21]. We present a uniform method containing and extending the latter results.

3.1 The Generic Problem

For the general discussion of the Dynamic Tree Expression Problem, we assume that we are given

i. a set P of N boolean *variables,* $p_1, \ldots, p_N$;

ii. a set I of *inference rules* of the form

$$p_i :- p_j p_k \text{ or } p_i :- p_j.$$

Here, juxtaposition of boolean variables is to be interpreted as logical AND, and "$:-$" is to be read as "if". In fact, the two types above are Horn clauses with one or two hypotheses, written in a PROLOG style notation: $(p_j \wedge p_k) \Rightarrow p_i$ and $p_j \Rightarrow p_i$, respectively. We note that the total length of I is polynomial in N.

iii. a distinguished subset $Z \subseteq P$ of *axioms.*

Definition 3.1 *Let (P, I, Z) be a system* as *above. The* minimal model *for (P, I, Z) is the* minimal *subset* $M \subseteq P$ *satisfying the following properties:*

(i) *the set of axioms, Z, is contained in M;*

(ii) *whenever the righthand side of an inference rule is satisfied by variables in M, then . the variable on the lefthand side is an element of M:*

$$\begin{array}{l} p_j, p_k \in M,\ p_i :- p_j p_k \in I \\ p_j \in M,\ p_i :- p_j \in I \end{array} \Rightarrow p_i \in M.$$

We say that *(P, I, Z) implies* some fact *p* if the boolean variable *p* is in the minimal model *M* for *(P, I, Z).* For each such *p,* there is a derivation or *proof tree:* This is a (rooted) tree whose internal vertices have one or two children, and whose vertices are labelled with elements in *P* such that these four properties are satisfied:

1. the labels of the leaves are in Z;

2. if a vertex labelled p_i has one child, with label p_j, then $p_i :- p_j$ is a rule in I;

3. if a vertex labelled p_i has two children, with labels p_j and p_k, then $p_i :- p_j p_k$ is a rule in I;

4. the label of the root vertex is *p.*

The *Dynamic Tree Expression Problem* consists in computing the minimal model for a given system *(P, I, Z),* or, formulated as a decision problem, in deciding whether, given *(P, I, Z)* and some $p \in P$, whether *p* is in the minimal model for *(P, I, Z).* The algorithm below can be used to solve DTEP on a PRAM.

algorithm *DTEP(N, P, I, Z);*
int N; set of boolean *P, Z;* set of inference rules I;
co *N* is the number of boolean variables in *P; Z* is the subset of *P* distinguished as axioms; *I* is a set of **Horn** clauses with at most two variables on the righthand side; *P* is implemented as an array of boolean variables; the algorithm sets to **true** exactly those elements of the array *P* corresponding to elements in the minimal model oc
begin
array $DI[1..N, 1..N]$;
co initialization oc
$P[i]$: = **true** for all $p_i \in Z$, else **false ;**
$DI[j, i]$ **:= true** for i = j and all $p_i :- p_j \in I$, else **false;**

do l times co l will be specified below oc
for i ∈ **(1,. . . ,** N} with *P[i]* = **false do in parallel**
if ((*P[j]* = *P[k]* = **true) A** $(p_i :- p_j p_k \in I)) \vee$ **((** $P[j] =$ **true)** $\wedge (DI[j,$ **i] = true))**

```
        then P[i] := true
        fi;
        if (P[k] = true) ∧ (p_i :- p_j p_k ∈ I) then DI[j, i] := true fi
    od;
    DI := DI · DI
  od
end DTEP.
```

We first remark about the intuition behind the array *DI* used in the DTEP algorithm. If entry (i, j) of this array is **true** then it is known that p_i implies p_j. This is certainly true at the beginning of the algorithm, by way of the initialization of *DI*. Whenever for an inference rule $p_i :- p_j p_k$ with two antecedents, one of them, say p_k, is known to be **true,** the rule can be simplified to $p_i :- p_j$. But this means that p_j implies p_i, as recorded in the assignment to $DI[j, i]$. Also, the net effect of squaring the matrix *DI* in the last step of the outer loop is that chains of implications are shortened. Note that entry (i, j) of the matrix becomes **true** through the squaring of the matrix only if there was a chain of implications from p_i to p_j before the squaring.

Lemma 3.1 *If p_i is in the minimal model M, and if there is a derivation tree for p_i in (P, I, Z) of size m then P[i] = true after at most $l = 2.41 \log m$ iterations of the outer loop of the DTEP algorithm. P[i] never becomes* **true** *for $p_i \notin M$.*

Proof: The proof is by induction on the number of iterations of the main loop. We use the following induction hypothesis:

> At the end of each iteration of the main loop, there is a derivation tree for p_i of size at most 3/4 of the size at the end of the previous iteration, allowing as inference rules the rules in I and the "direct implications" as given by the current values of the elements of *DI,* and as axioms the p_j with *P[j]* currently **true.**

This induction hypothesis is trivially satisfied before the first iteration of the loop. For the r-th iteration, let $m = |T|$ be the size of a derivation tree T for p_i, using the current axioms and derivation rules. **We** distinguish two cases:

1. *T* has at least *m/4* leaves: since the parallel loop in the DTEP algorithm set *P[k]* = **true** if the children of a vertex labelled p_k are all leaves, this case is trivial.

2. Let r_k be the number of maximal chains in *T* with *k* internal vertices, and let *b* be the number of leaves of *T,* both at the start of the loop. A chain in *T* is part of a branch with all internal vertices having degree exactly 2. Then we have:

$$m = 2b - 1 + \sum_{k \geq 1} k \cdot r_k;$$

$$m' \leq b - 1 + \sum_{k \geq 1} \lceil k/2 \rceil \, r_k$$

$$\begin{aligned} &\leq b - 1 + \frac{1}{2}(\sum_{k\geq 1} kr_k + \sum_{k\geq 1} r_k) \\ &\leq b - 1 + \frac{1}{2}(m + 1 - 2b) + \frac{1}{2} \cdot 2b \\ &< \frac{3}{4}m \ . \end{aligned}$$

Here, m' is the size of the derivation tree after execution of the loop, using the current set of axioms and inferences. The first inequality follows since squaring of the matrix DI halves the length of all chains. The third inequality stems from the observation that in a tree with b leaves, the number of (maximal) chains can be at most $2b - 1$ as can be seen by assigning each such chain to its lower endpoint.

There is a trivial derivation tree for p_i after at most $\log_{4/3} m \leq 2.41 \log m$ (note that all logarithms whose base is not mentioned explicitly, are base 2). □

Theorem 1 Let *(P, I, Z) be a derivation system with the property that each p in the minimal model M has a derivation tree of size*

$$\leq 2^{\log^c N}.$$

Then there is an $\mathcal{NC}$-algorithm to compute M.

Proof: It follows immediately from the above Lemma that under the stated conditions the DTEP algorithm runs in polylogarithmic time. More precisely, for the bound on the size of derivation trees given in the Theorem, the running time of the algorithm is

$$O(\log^{c+1} N),$$

and it uses N^3 processors. □

3.2 Applications of DTEP

Longest Common Substrings

As a very simple example of a DTEP application we consider the *longest common substring* problem: given two strings $a_1 \ldots a_n$ and $b_1 \ldots b_m$ over some alphabet Σ, we are supposed to find a longest common substring $a_i \ldots a_{i+l} = b_j \ldots b_{j+l}$.

To obtain a DTEP formulation of this problem, we introduce variables $p_{i,j,l}$ whose intended meaning is:

$$p_{i,j,l} \text{ is true if } a_i \ldots a_{i+l-1} = b_j \ldots b_{j+l-1}.$$

Thus, we have the following axioms:

$$p_{i,j,1} \text{ iff } a_i = b_j;$$

and these inference rules:

$$p_{i,j,l} :- p_{i,j,\lceil l/2 \rceil} P_{i+\lceil l/2 \rceil, j+\lceil l/2 \rceil, \lfloor l/2 \rfloor}.$$

An easy induction on the length l of a common substring shows that $p_{i,j,l}$ is true in the minimal model for the above derivation system if and only if the substring of length l starting at position i in the first string is equal to the substring of the same length at position j in the second string. A similar induction can be used to show that for every common substring of length l, there is a derivation tree of size linear in l in the above derivation system. These two observations together establish the longest common substring problem as an instance of DTEP.

Other Applications

It is possible to rephrase a number of other problems as instances of the *Dynamic Tree Expression Problem.* A few examples are contained in the following list:

- transitive closure in graphs and digraphs
- the non-uniform word problem for context-free languages
- the circuit value problem for planar monotone circuits
- DATALOG programs with the polynomial fringe *property*
- the complexity class ALT(log n, expolylog n); these are the problems recognizable by log-space bounded Alternating Turing machines with a bound of $2^{\log^c n}$ for the size of their computation tree, for some constant c > 0.

For the DATALOG example, we refer the interested reader to [21]. We briefly discuss the third and the last example in the above list.

The Planar Monotone Circuit Value Problem

A *circuit* is a directed acyclic graph (dag). Its vertices are called *gates,* and the arcs correspond to wires going from outputs of gates to inputs. The gates are either input/output gates connecting the circuit to its environment, or they are combinational mapping the values on the input wires of the gate to values on the output wires, according to the function represented by the gate. In a *monotone* circuit, all combinational gates are AND or OR gates. Therefore, the output of the circuit, as a function of the inputs, is a monotone function. **A** planar circuit is a circuit whose dag can be laid out in the plane without any wires crossing one another, and with the inputs and output(s) of the circuit on the outer face.

We assume that circuits are given as follows. The description of a circuit with n gates is a sequence $\beta_0, \ldots, \beta_{n-1}$. Each gate β_i is

1. O-INPUT or 1-INPUT (also abbreviated as 0 and 1, respectively); or

2. AND(β_j, β_k), OR(β_j, β_k), or NOT(β_j), where j, $k < i$; in this case, the gate computes the logical function indicated by its name, with the values on the input wires to the gate as arguments, and puts this function value on all of the gate's output wires.

3. The output wire of β_{n-1} carries the output of the circuit.

The *circuit value problem* (CVP) consists in computing the output of a circuit, given its description. It is well known that CVP is complete for $\mathcal{P}$ under log-space reductions, and hence is a "hardest" problem for the class of polynomial time algorithms [9]. Even if we restrict ourselves to monotone circuits (allowing only AND and OR gates), or to planar circuits, the corresponding restricted CVP remains P-complete [5]. However, the circuit value problem can be solved efficiently in parallel for circuits which are monotone and planar [3], [6]. While the original solutions were indirect and technically quite involved, the DTEP paradigm gives us a relatively simple approach.

For ease of presentation, we assume that the description $\beta_0, \ldots, \beta(n-1)$ of a planar monotone circuit satisfies the following additional properties:

- The circuit is arranged in *layers;* the first layer is constituted by the inputs to the circuit; the last layer consists of the output of the circuit; for every wire in the circuit, there is a (directed) path from an input to the output of the circuit, containing that wire.
- All wires run between adjacent layers, without crossing one another.
- All gates on even layers are AND gates with at least one input.
- All gates on odd layers (except the first) are OR gates.

These restrictions are not essential. Given a general type description of a circuit, there are well-known NC-algorithms to test whether it represents a monotone and planar circuit, and to transform it into a description of a functionally equivalent circuit satisfying the restrictions listed above.

To obtain a DTEP formulation, it is helpful to look at intervals of gates. Formally, let the triple *(1, i, j)* denote the interval from the ith through the jth gate (in the order in which they appear in the circuit description) on the lth layer of the circuit. We shall introduce boolean variables $p(l, i, j)$ with the intended meaning

$p_{l,i,j}$ is **true** iff the output of every gate in (l, i, j) is **true.**

We simply observe that an interval of AND-gates has outputs all **true** if and only if all inputs to this interval of gates are **true,** and that these inputs come from an interval of gates on the next lower layer. For intervals of OR-gates, the situation is a bit more complicated since in order to obtain a **true** output from an OR-gate, only one of its inputs needs to be **true.** We must therefore be able to break intervals of **true** OR-gates down into smaller intervals which are **true** because a contiguous interval of gates on the next

lower layer is **true.** Because wires don't cross in a planar layout, such a partition is always possible.

More formally, the inference rules for **a** given instance of the planar monotone circuit value problem are:

1. $p_{l,i,j} :- p_{l-1,i',j'}$ for every (nonempty) interval of AND-gates; here, i' denotes the first input of the first gate, and j' the last input of the last gate in the interval;

2. $p_{l,i,j} :- p_{l,i,k}p_{l,k+1,j}$, for all $i \leq k < j$; and

3. $p_{l,i,j} :- p_{l-1,i',j'}$, for all layers l of OR-gates and all pairs (i', j') where i' $\leq$ j' and $\beta_{i'}$ is an input of β_i, and $\beta_{j'}$ is an input of β_j.

The axioms are given by all intervals of **true** inputs to the circuit.

It is easy to see that the total length of the representation for the axioms and for all inference rules is polynomial in n. A straightforward induction, using the planarity of the circuit as outlined above, **also** shows that $p_{l,i,i}$ is **true** in the minimal model if and only if the ith gate on layer l has output **true** in the circuit.

Suppose $p_{l,i,j}$ is in the minimal model. To see that there is a linear size derivation tree for $p_{l,i,j}$ we consider the sub-circuit consisting of all gates from which **a** gate in (l, i, j) is reachable. Whenever we use an inference rule of type 1 or of type 3 in a derivation tree for $p_{l,i,j}$ we charge the vertex in the derivation tree (corresponding to the left-hand side of the rule) to the interval ($l, i,$ j). We observe that we need to use an inference rule of type 2 for an OR-gate on some layer *1'* only if the interval of gates on layer $l' - 1$ feeding into this interval contains more than one maximal sub-interval of **true** gates. In this case, we can split the interval (l', i, j) into sub-intervals *(I',* i, *k)* and (l', k + 1, j) in such a way that one of the inputs to gate *k* on layer *1'* is **false.** We charge the left-hand side of the type 2 inference rule to the interval of **false** gates on layer *1'* $-$ 1 containing this input. Because of the planarity of the circuit, rules used to derive $p_{l',i,k}$ and $p_{l',k+1,j}$ are charged to disjoint sets of intervals. Therefore, the number of maximal intervals of gates with the same output value is an upper bound on the size of a smallest derivation tree for any element in the minimal model. We conclude

Theorem 2 *There is an NC algorithm for the planar monotone circuit value problem.*

□

Tree Size Bounded Alternating Turing Machines

Alternating Turing Machines are a generalization of standard (nondeterministic) Turing Machines [8], **[l].** They are interesting because their time complexity classes closely correspond to (standard) space complexity classes, and because their computations directly correspond to first order formulas in prenex form with alternating quantifiers.

For an informal description of Alternating Turing Machines (ATM's), we assume that the reader is familiar with the concept of standard (nondeterministic) Turing machines

[8]. An ATM consists of a (read-only) input tape and some number of work or scratch tapes, as in the standard model, as well as a finite control. The states of the finite control, however, are partitioned into *normal, existential,* and *universal* states. Existential and universal states each have two successor states. Normal states *can* be *accepting, rejecting,* or *non-final.* An *instantaneous description* or id of an ATM consists of a string giving the state of the finite control of the ATM, the position of the read head on its input tape, and head positions on and contents of the work tapes. It should be clear that, given the input to the machine, an id completely determines the state of the machine.

A computation of an ATM can be viewed as follows. In any state, the machine performs whatever action is prescribed by the state it is in. If the state is a normal non-final state, the finite control then simply goes to its unique successor state. Accepting or rejecting states have no successors. If the state is existential or universal, the machine clones itself into as many copies as there are successor states, with each clone starting with an id identical to that of the cloning id, except that the state of the cloning id is replaced by that of the successor state. The cloning machine disappears. It is best to view the computation of an ATM as a tree, whose nodes correspond to id's of the computation, in such a way that children of a node in the tree represent successor id's.

Given this interpretation, each computation of an ATM can be mapped into a (binary) tree. This tree need not necessarily be finite, since there can be non-terminating branches of the computation. To define whether an ATM *accepts,* or, dually, *rejects* some input, we look-at the corresponding computation tree as defined above. A *leaf* in this tree is called *accepting* iff the state in its id is accepting. An internal node whose id contains a normal state is accepting iff its (unique) descendent in the tree is accepting. An internal node with an existential state is accepting iff at least one of its immediate descendents is accepting, and a node with a universal state iff all of its immediate descendents are accepting. An id of the ATM is called *accepting* if it is attached to an accepting node in the computation tree. An ATM is said to accept its input if and only if the root of the corresponding computation tree is accepting by the above definition, or, equivalently, if the initial configuration of the machine is accepting.

To cast ATM computations into the DTEP framework we note that if the space used by the ATM on its worktapes is bounded by $O(\log n)$ then the number of distinct id's for the machine is bounded by some polynomial in n. Assume that $id_0, \ldots, id_m$ is an (efficient) enumeration of these id's. Our instance of DTEP will have boolean variables p_i, for $i = 0, \ldots, m$, with the intended meaning

p_i is **true** iff id_i is accepting.

The axioms are given by those id's containing accepting states, and we will have inference rules

- $p_i :- p_j p_k$ iff p_i contains a universal state with two successor states, and id_j and id_k are the two immediate successor id's of id_i; and
- $p_i :- p_j$ in all other cases where id_j is a (the) immediate successor id of id_i.

$$x_4 = x_1 + x_2 \quad x_5 = x_4 \text{ x } x4$$
$$x_6 = x_5 \text{ x } x4 \quad x_7 = x_1 \text{ x } x_1$$
$$x_8 = x_3 \times x_7 \quad x_9 = x_6 \times x_8$$
$$x_{10} = x_4 \times x_9$$

Figure **1:** Straightline program for polynomial ((xi + x_2)^3 + $x_1^2 x_3$)(x_1 + x_2)

A straightforward induction shows that there is an exact correspondence between derivation trees for the p_i corresponding to the initial id of the ATM, and certain subtrees of the ATM's computation tree. These subtrees are obtained by removing, for each node whose id contains an existential state, all but one of the children of that node, together with their subtrees. Also, the leftover child has to be an accepting node.

Since the computation tree of the ATM is of polynomial size, so is any derivation tree obtained by the above construction. This concludes the (informal) proof for

Theorem *3 Problems* in *the complexity class ALT(l*0gn, expolylog *n) are instances of the Dynamic Tree Expression Problem. In particular, they are members of NC.*

□

4 Application to Algebraic Straightline Programs

Straightline programs consist of a sequence of assignment statements executed in order. Each variable referenced in a statement is either an input variable, or it has been assigned a value in an earlier assignment. In *algebraic straightline programs,* the expressions on the righthand side of the assignment are basic unary or binary expressions involving the arithmetic operators +, −, x, and /, as well as constants and input or computed variables. Each non-input variable is assigned a value in exactly one statement in the straightline program. In what follows, we are only concerned with straightline programs containing the operators +, −, and x. Any such straightline program computes some (multivariate) polynomial of the input variables. Conversely, every multivariate polynomial can be expressed as an algebraic straightline program.

-A straightline program corresponds, in a natural way, to a directed acyclic *graph* or *dag.* In this dag, the vertices correspond to the assignment statements, the sources of the dag (i. e., vertices of indegree zero) correspond to the input variables of the program, and arcs signify the use of variables defined in one statement by some other statement. We define the size of a straightline program to be its number of assignment statements. Figure 1 shows a straightline program for the polynomial $((x_1 + x_2)^3 + x_1^2 x_3)(x_1 + x_2)$ with indeterminates x_1, x_2, and x_3, and Figure 2 gives the corresponding computation dag. In Figure 2, all edges represent arcs directed upward.

We assume without loss of generality that the operation in the last assignment in a straightline program is a multiplication.

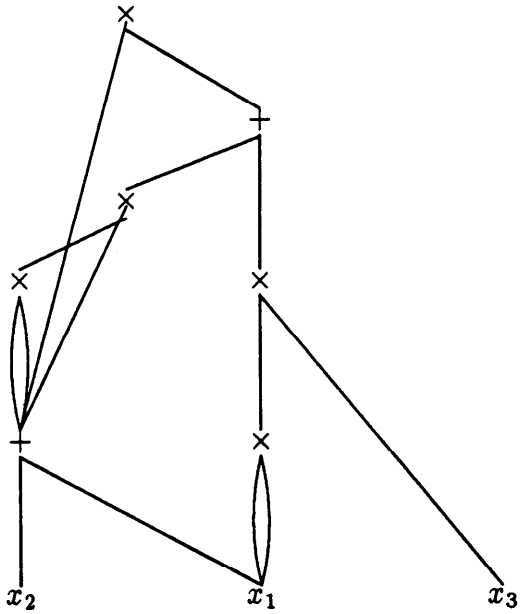


Figure 2: Computation dag for straightline program in Figure 1

To make straightline programs more uniform and easier to deal with, we first wish to transform them into an (algebraically equivalent) sequence of *bilinear forms* of the form

$$y_i = (\sum_{j<i} c_{ij}^{(l)} y_j) \cdot (\sum_{j<i} c_{ij}^{(r)} y_j).$$

Again, the y_j are either input variables or assigned a value in an earlier assignment. In fact, each non-input variable y_i is in exact correspondence with some x_{k_i} in the original straightline program which is the result of a multiplication operation. Assuming that $i = k_i$ for the input variables, the coefficients $c_{ij}^{(l)}$ (resp., $c_{ij}^{(r)}$) give the number of distinct paths from x_{k_j} to the left (resp., right) multiplicand of x_{k_i} in the dag belonging to the original straightline program. These coefficients can be obtained in a straightforward way by repeatedly squaring a matrix *A* obtained as follows. We first construct, from the computation dag for the straightline program, an auxiliary dag by splitting every multiplication node v into three nodes, v^1, v^2, and v^3: v^1 receives the arc from the left multiplicand, v^2 the arc from the right multiplicand. All arcs originally leaving v are reattached to v^3. In the resulting digraph, the original input nodes and the type 3 nodes are sources, the type 1 and type 2 nodes are the sinks. The matrix *A* mentioned above is the adjacency matrix of this digraph, after we attach a loop (directed cycle of length one) to every sink. Further details are left to the reader. The coefficients in the bilinear forms can be computed on a PRAM in time $O(\log^2 n)$, using *M(n)* processors. Here, *n* is the size of the original straightline program, and $M(n) \leq n^3$ is the number of processors required to perform matrix multiplication in time O(logn).

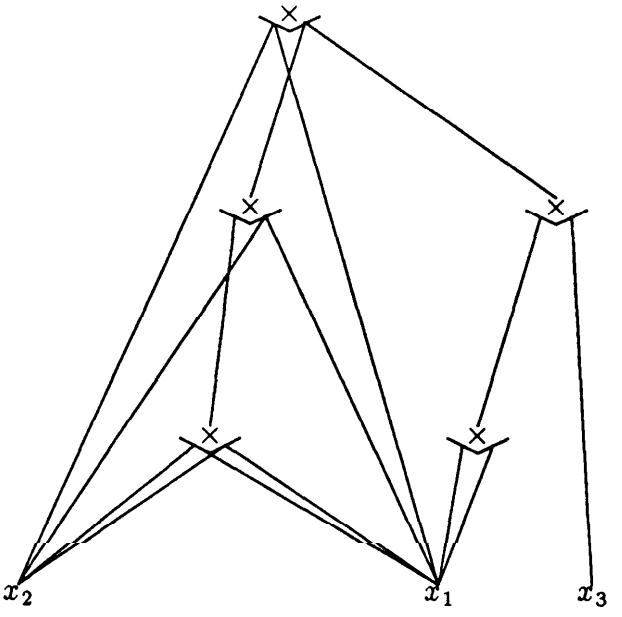


Figure 3: Graph of bilinear straightline program

Figure 3 shows a graphical representation of the transformed "bilinear straightline program" for the straightline program resp. computation dag given in Figures **1** and **2.** Each bilinear form is represented by a flat 'V', with the terms in the first (second) factor given by the arcs ending in the left (right) arm of the 'V'. The coefficient belonging to a term is attached to the arc unless it is one or zero. In the latter case, no arc is drawn at all.

For the digraph B of a bilinear straightline program, we can define operations analogous to the DTEP operations. The boolean variables are replaced by pairs (c_i, v_i) for every node i in B. The first element, c_i, is a boolean indicating whether the value of the polynomial corresponding to vertex i has been computed; if c_i is **true** then v_i holds this value. Initially, the values of all the input variables are known. We call a node of B a l-node iff it is not a sink (*i.* e. a node of outdegree zero) and the value of exactly one of its factors is known (i.e. the values of all nodes with arcs to the corresponding "leg" have been computed). All-other nodes whose value is still unknown are called *2-nodes*.

We also have several matrices describing the arcs in the graph, and the coefficients attached to them. The set of arcs and coefficients, and consequently the entries in these matrices, change during the course of the algorithm.

- the $(i,$ j)th entry of $C^{(l)}$ contains the coefficient of the arc from the ith node of B to the *left leg* of the jth node, for all nodes i and j such that j is a 2-node; the other entries are zero;
- $C^{(r)}$ is defined correspondingly for the righthand factors;

- the (i, j) entry of the matrix D, for j a l-node, is the coefficient of the arc from node i to node j;
- the matrix $\bar{C}$ is a projection matrix; all its entries are zero except for those on the diagonal positions corresponding to 2-nodes.

Each phase of the adapted DTEP algorithm consists of the following steps:

do one phase of adapted DTEP oc
for all i with c_i = **false** and both factors known **do in parallel**
compute v_i and set c_i = **true;**
for all nodes i which just became new l-nodes **do in parallel**
multiply the coefficients of all arcs entering the unknown "leg" by the other, known factor;
update all matrices so as to reflect the changes in the sets of l-nodes and %-nodes;
$C^{(l)} := (\bar{C} + D) \cdot C^{(l)}$; $C^{(r)} := (\bar{C} + D) \cdot C^{(r)}$;
$D := (\bar{C} + D) \cdot D$;
update B to correspond to the newly computed matrices;
co end of phase oc

It is straightforward to verify that the bilinear program given by the modified graph after execution of one phase computes the same values as the original bilinear program. Also, one phase can be executed on a PRAM in time $O(\log n)$ employing M(n) processors, where M(n) $\leq n^3$ is the number of processors required to multiply two n x n matrices in 0(log n) parallel steps.

To obtain a bound on the required number of phases, it is not sufficient to unfold the dag B into a tree and then use our general results about DTEP since there are easy examples where the unfolded tree is of exponential size. However, the following worst-term argument works.

Define, for each node in the graph of a bilinear straightline program, its *formal degree* inductively as follows: The original input variables have formal degree 1, and each other variable has a formal degree equal to the sum of the formal degrees of its two factors. Here, the formal degree of a factor is defined to be the maximum of the formal degrees of all its terms.

For every factor of a non-input variable y_i in a bilinear straightline program, we define a *worst term*, derived from the execution of the above algorithm: determine, for the corresponding "leg" of the node i in the sequence of graphs generated by the algorithm, which other node j with an arc to that leg is the latest to have its value computed. If j has an arc to i in the first graph of the sequence, y_j becomes the worst term for the factor of y_i under consideration, and we distinguish the arc from j to i. Otherwise, distinguish any path from j to the appropriate leg of i in the first graph of the sequence. Every node on this path corresponds to the worst term of the appropriate factor of the next node on the path.

Consider the subgraph B' induced by the distinguished arcs. Assume without loss of generality that B' has only one sink. Clearly, B' computes some monomial. If we unfold

B' into a tree T then T's size is exactly twice the formal degree of this monomial, minus one. The reason is that all non-leaf nodes in T are multiplication nodes, with exactly one term per factor.

A simple induction shows that the adapted DTEP algorithm computes the values of nodes in T in exactly the same phases as it computes them when run on B, the full graph. The analysis of the general DTEP algorithm applies to the adapted algorithm, executed on T. We obtain that the adapted DTEP algorithm computes the value of each y_i in a number of phases proportional to the logarithm of the formal degree of y_i. Since the initial transformation to a bilinear straightline program is basically a transitive closure computation, we obtain the

Theorem 4 *Suppose S is a straightline program containing n variables and computing a polynomial of formal degree d. The adapted DTEP algorithms computes the value of the polynomial given by S, at a specified point, in time* $O(\log n(\log n + \log d))$ using $M(n)$ *processors on a unit cost PRAM.*

□

Thus, polynomials given by straightline programs can be evaluated by an NC algorithm as long as their formal degree is expolylog in n. This result was originally obtained in [13]. Our presentation is intended to give evidence for the wide applicability of the DTEP paradigm, and maybe to supply a somewhat simpler, less technical proof.

5 Conclusion

The DTEP paradigm establishes an interesting connection between fast parallel computation and small proof or derivation trees. In more technical terms, the instances of DTEP in $\mathcal{NC}$ are characterized as ALT(log n, expolylogn) or the set of problems recognizable by log space bounded alternating Turing machines with "expolylogarithmically" bounded computation trees. DTEP is not necessarily intended to provide optimal parallel algorithms for a given problem. Instead, its primary function is to provide a convenient, high level way for the specification of an efficient and highly portable parallel algorithm. The input to DTEP is in a format very similar to logic programs, rather declarative than procedural, and thus allows to abstract away many peculiarities of any underlying parallel architecture.

The DTEP algorithm can be efficiently implemented on various "real world" parallel architectures, like binary hypercubes, butterfly networks, shuffle-exchange based networks, and multi-dimensional meshes of trees [12]. The algorithm can also be tailored and made more efficient (both in terms of time and number of processors) for more restricted instances of the dynamic tree expression problem. Some of these optimizations are also discussed in [12].

We have also shown how to extend the basic DTEP paradigm to problems over algebraic domains, in particular the parallel evaluation of algebraic computation dags, or straightline programs. Further interesting extensions include the possibility of handling non Horn clauses in the basic formulation of DTEP, along the lines discussed in [21].

Another interesting line of research is the characterization of more classes of problems for DTEP in independent terms, in particular the derivation of conditions for the existence of small proof trees.

References

[1] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J.ACM*, 28(1):114–133, 1981.

[2] R. Cole. Parallel merge sort. In *Proceedings of the 27th Ann. IEEE Symposium on Foundations of Computer Science (Toronto, Canada),* pages 511-516, 1986.

[3] P. Dymond. *Simultaneous resource bounds and parallel computation.* Ph.D. Thesis, Department of Computer Science, University of Toronto, 1980.

[4] S. Fortune and J. Wyllie. Parallelism in random access machines. In Proceedings *Of the 10th Ann. ACM Symposium on Theory of Computing (San Diego, CA),* pages 114–118, 1978.

[5] L. Goldschlager . The monotone and planar circuit value problems are log-space complete for $\mathcal{P}$. *SIGACT News,* 9(2):25–29, 1977.

[6] L. Goldschlager. A space efficient algorithm for the monotone planar circuit value problem. *Information Processing Letters,* 10(1):25–27, 1980.

[7] P. Hochschild, E. Mayr, and A. Siegel. Techniques for solving graph problems in parallel environments. In *Proceedings of the 24th Ann. IEEE Symposium on Foundations of Computer Science (Tucson, AZ),* pages 351-359, 1983.

[8] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[9] R. Ladner. The circuit value problem is log-space complete for $\mathcal{P}$. *SIGACT News,* 7(1):583–590, 1975.

[10] R. Ladner and M. Fischer. Parallel prefix computation. *J.ACM*, 27(4):831–838, 1980.

[11] R. Lipton and J. Valdes. Census functions: An approach to VLSI upper bounds (preliminary version). In *Proceedings of the 22nd Ann. IEEE Symposium on Foundations of Computer Science (Nashville, TN),* pages 13-22, 1981.

[12] E. Mayr and G. Plaxton. *Network Implementations of the D TEP Algorithm.* Technical Report, Department of Computer Science, Stanford University, 1987.

[13] G. Miller, V. Ramachandran, and E. Kaltofen. *Efficient parallel evaluation of straight-line code and arithmetic circuits.* Technical Report TR-86-211, Computer Science Dept., USC, 1986.

[14] G. Miller and J. Reif. Parallel tree contraction and its application. In *Proceedings of the 26th Ann. IEEE Symposium on Foundations of Computer Science (Portland, OR),* pages 478-489, 1985.

[15] G. Pfister. *The architecture of the IBM research parallel processor prototype (RP3).* Technical Report RC 11210 Computer Science, IBM Yorktown Heights, 1985.

[16] N. Pippenger. On simultaneous resource bounds. In Proceedings *of the 20th Ann. IEEE Symposium on Foundations of Computer Science (San Juan, PR),* pages 307–311, 1979.

[17] F. Preparata and J. Vuillemin. The cube-connected-cycles: A versatile network for parallel computation. In Proceedings *of the 20th Ann. IEEE Symposium on Foundations of Computer Science (San Juan, PR),* pages 140-147, 1979.

[18] W. Ruzzo. Tree-size bounded alternation. In Proceedings *of the 11 th Ann. ACM Symposium on Theory of Computing (Atlanta, GA),* pages 352-359, 1979.

[19] J. Schwartz. Ultracomputers. *ACM Transactions on Programming Languages and Systems,* 2(4):484–521, 1980.

[20] C. Seitz. The cosmic cube. *CACM,* 28(1):22–33, 1985.

[21] J. Ullman and A. van Gelder. Parallel complexity of logical query programs. In *Proceedings of the 27th Ann. IEEE Symposium on Foundations of Computer Science* (Toronto, Canada), pages 438454, 1986.