Information and Computation **143**, 231–250 (1998)
Article No. IC982702

A Constant-Space Sequential Model of Computation for First-Order Logic*

Steven Lindell†

Haverford, Pennsylvania 19041-1392
E-mail: slindell@haverford.edu

We define and justify a natural sequential model of computation with a constant amount of read/write work space, despite unlimited (polynomial) access to read-only input and write-only output. The model is deterministic, uniform, and sequential. The constant work space is modeled by a finite number of destructively read boolean variables, assignable by formulas over the canonical boolean operations. We show that computation on this model is equivalent to expressibility in first-order logic, giving a duality between (read-once) constant-space serial algorithms and constant-time parallel algorithms.

0. INTRODUCTION

Summary

Problems computable in constant time on a uniform parallel model of computation (a type of PRAM) have been elegantly characterized as those expressible in first-order logic (FO) on binary strings [I]. It is also known that FO is identified with LH, the logtime alternation hierarchy based on random-access Turing machines [BIS]. We provide an additional correspondence between FO and those problems computable in constant space on a deterministic sequential model of computation.

It is well known that Turing machines operating in constant space are equivalent to finite automata and hence accept only the class of regular languages. The key to capturing the logtime alternation hierarchy as a space-bounded complexity class is a very careful measurement of work space in a machine. Ordinarily, read-only input and write-only output are not considered part of the read/write work space. This is an essential concept for defining the complexity class L (logarithmic-space). We go somewhat farther in our model and do not include in the work space any storage mechanism required to access the input or output, be it memory addressing or tape scanning. By making the access scheme oblivious, we are careful not to let

* A preliminary version of this paper appeared in "*Logic and Computational Complexity*" (Daniel Leivant, Ed.), Lecture notes in Computer Science No. 960, pp. 447–462, Springer-Verlag, Berlin, 1995.
† Partially supported by NSF grant CCR-9403447 and the John C. Whitehead faculty research fund at Haverford College.

0890-5401/98 $25.00

the machine cheat by using the memory addresses or head positions as read/write storage.

Furthermore, we take the additional step of separating the flow of control of the machine from the computation it is performing. Specifically, we imagine a machine controlled by a simple programming language with: a finite set of read/write boolean variables; the operations AND, OR, NOT; composition of program statements; and a strict form of definite loops. No conditionals are allowed in the programming language (if … then, or while … repeat) to ensure the oblivious nature of the computation. In addition, we impose a read-once (destructive read) condition that prevents a read/write boolean variable from being read more than once without an intervening write. No such restriction applies to input or output however.

The following is representative of our main theorem:

THEOREM. *A query on binary strings is first-order definable if and only if it is computable by a constant-space read-once serial algorithm.*

Motivation

Classically, when defining sub-linear space on a Turing machine, we resort to an off-line model which separates read-only input and write-only output from the read/write work tape. In this way we can get robust definitions of DSPACE ($\log n$) and above. And although log-space transducers have proved to be a useful reducibility between problems, a finer notion of reduction based on first-order translations has led to some illuminating results: a very restricted version of the Berman–Hartmanis conjecture [ABI]; and a very deep result concerning the recursive enumerability of the polytime queries [D′]. It has also been shown that first-order logic provides a robust notion of uniformity for the study of the fine structure of low-level circuit complexity classes [BIS], and the corresponding first-order reductions have been shown equivalent in [AG] to a much earlier notion of logspace rudimentary reductions. Furthermore, first-order translations are based on the classical notion of interpretation as spelled out in [E], and serve as excellent reductions which preserve the completeness of well-known NP-problems [D] as well as newer ones (the boolean formula value problem being complete for ALOGTIME) [B].

Finite-state transducers fail in this capacity because they cannot do the arithmetic needed to convert binary input from one simple form into another (like reversing a string), nor can they provide polynomial magnification (required for the existence of complete problems). This happens in any standard off-line model with space below $O(\log n)$.

Our sequential model of computation is able to lift these limitations while still operating under a form of constant-space constraint. This is because our model permits multiple uni-directional heads which can re-scan the tape in definite loops (dependent on input length), but forbids head movement which is non-oblivious (i.e., dependent on input contents). We will measure the actual amount of read/write work space in a very careful fashion. In particular, our model does not

even count any space used to access the input, whether it be a head scanning a tape, or an address into random-access memory.[1] The reasoning for this is intuitive: if the input (output) tape is read-only (write-only), and the access to the tape is oblivious, then the machine cannot use the tape head as a read/write storage mechanism, so that space does not count. Clearly this intuition, if correct, extends to any fixed number of heads. If access to the tape is not oblivious, and we allow two-way multihead finite automata, then all of logspace is achieved ([G], p. 51). In fact, our model will be explained in terms of a programming language, much the same as the presentation of primitive recursion in [G, p. 20], and can be compared with the uniform constant-width circuits of [BI, Section 6], which gives a corresponding characterization of uniform-NC^1. Also, [C] has given machine-independent algebraic characterizations of AC^0 and various other small complexity classes using sequential operations.

Overview

Section 1 provides a brief review of first-order definable queries, including examples. Of particular importance is the discussion of binary string structures and the special numerical predicates for arithmetic. Section 2 defines and justifies the constant-space model of sequential computation that this paper introduces, comparing it with the classical definition of finite-state automata. Section 3 illustrates this model with two contrasting bit-serial examples: addition and parity. Section 4 contains the main result, mentioned above, and its proof. The remainder of the paper, Section 5, concludes by describing possible directions for future research, including extensions to TC^0, and examination of a serial multiplication algorithm.

1. BACKGROUND

First-Order Queries

One way of mathematically studying the computational complexity of combinatorial problems is to directly examine how difficult it is to define them. Instead of measuring asymptotic resources (such as time or space) required to compute a problem, one can classify problems as to the power of the logic required to express their solution. Specifically, an input instance is a *finite relational structure*,

$$\langle A, R_1, ..., R_k, c_1, ..., c_l \rangle,$$

consisting of a finite set A, called the *domain*, together with *relations* R_i (each of a specified arity on A) and *constants* c_j (individual elements of A). The output is determined by a *query*, a global relation across structures of the same type (signature), mapping each one to a relation (of fixed-arity) on the structure. One

[1] However, it will be preferable to use a cursor to mark head position, since this method of memory access appears aesthetically more sequential, as opposed to RAM which involves an implicit use of parallelism (address decoding).

of the simplest "languages" for expressing queries is that of *first-order* logic. Formulas in first-order logic permit: individual variables interpreted as ranging over the domain; constant symbols for each constant c_j, predicate symbols for each relation R_i and equality $(=)$; the Boolean connectives $\wedge$, $\vee$, and $\neg$; and quantification of the variables. We use FO to denote the class of all queries determined by such first-order formulas. (For further background, see [E]).

Simple Graphs

One of the easiest and most familiar examples of finite structures is the class of directed graphs—all structures having a binary edge relation E over a finite domain of vertices V:

$$G = \langle V, E \rangle \qquad E \subseteq V^2.$$

The problem of determining if a graph is *simple* (no self-loops, all edges undirected) is an example of a graph property, or boolean query of arity 0 which is expressible as a first-order sentence:

$$G = \langle V, E \rangle \qquad \text{is } \textit{simple} \text{ iff} \quad G \models \theta, \qquad \text{where}$$
$$\theta \equiv \neg(\exists x)[E(x, x)] \wedge (\forall y)(\forall z)[E(y, z) \rightarrow E(z, y)].$$

The first part of θ says that no vertex has an edge to itself, and the second part says that if there is an edge from one vertex to another, then there must be an edge going in the opposite direction.

Linear Orderings

Another example is the problem of determining if a binary relation constitutes a total linear *order* of the vertices. We say

$$B = \langle A, < \rangle \qquad \text{is } \textit{ordered} \text{ iff} \quad B \models \psi,$$

where ψ is the conjunction of the following universally quantified axioms:

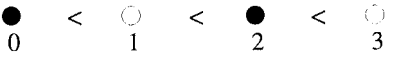
$$\neg(x < x) \qquad \text{(irreflexivity)}$$
$$(x \neq y) \rightarrow [(x < y) \vee (y < x)] \qquad \text{(totality)}$$
$$[(x < y) \wedge (y < z)] \rightarrow (x < z) \qquad \text{(transitivity)}.$$

Binary String Structures

Ordering the Positions

It seems necessary to work on ordered structures to express computation, particularly with regard to the contents of the input as it is presented to a machine.

Modern digital computers use binary strings for I/O, and we can represent these in the form $B = \langle \{0, 1, ..., n-1\}, <, U \rangle$, where the domain is the set of positions in the string, $<$ orders these positions from left to right in the usual fashion $0 < 1 < \cdots < n-1$, and U indicates where the 1's and 0's are by true and false, respectively. For instance, the binary string 1010 is represented by the structure $\langle \{0, 1, 2, 3\}, <, \{0, 2\} \rangle$, with $0 < 1 < 2 < 3$.



In the figure above, closed circles indicate where U is true and open circles where U is false.

In general, given a binary string $w \in \{0, 1\}^*$, we create a canonical structure for it,

$$\langle |w|, <, \{i : w_i = 1\} \rangle,$$

where $|w| = \{0, 1, ..., |w| - 1\}$ is the length of w, and w_i is the ith bit of w.

Adding Arithmetic

To capture accurately fine notions in resource-bounded computation (such as parallel time) appears to require, in addition to the ordering, a method whereby the binary input to a machine can be accessed effectively [I]. For this purpose (what might be called address arithmetic) it suffices to have a special predicate which, for each domain element i, gives the location of 1's and 0's in its binary representation:

$$\text{bit}(i, j)$$
$$\Leftrightarrow$$
the jth position in the binary representation of i is a 1.

For instance, bit(5, 1) is false, since $5 = (101)_2$, and there's a 0 in the first position (the rightmost bit is treated as the zeroeth position). This leads us to the following definition.

DEFINITION. For each $w \in \{0, 1\}^*$, define the *binary string structure* for w to be

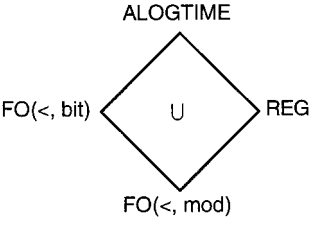
$$A_w = \langle |w|, <, \text{bit}, \{i : w_i = 1\} \rangle.$$

We distinguish the correspondingly augmented class of first-order queries by the notation FO($<$, bit), indicating that $<$ and bit are assumed to be "givens" in the same way $=$ is taken for granted. See [L] for a discussion of the logical importance of the bit predicate and its connection with arithmetic. Also, see [DDLW] for the surprising new result that FO($<$, bit) = FO(bit).

2. THE MACHINE MODEL

Comparison with Regular Languages

Before discussing the machine model, it will be instructive to compare the computational complexity of FO($<$, bit) with the more familiar regular languages, denoted REG, which are those recognized by constant-space Turing machines. First note that both FO($<$, bit) and REG are strictly contained in ALOGTIME ($=$uniform-NC^1). A classical result is that the regular languages, viewed as collections of binary string structures without bit, are precisely those definable in monadic second-order logic (see [S]). An equally important observation is the fact that first-order logic on binary string structures without bit corresponds exactly to the star-free fragment of REG. In fact, FO($<$, bit) $\cap$ REG is equal to the class of first-order definable queries on binary strings with order, together with the unary numerical predicates $M_k = \{m \cdot k : m = 0, 1, ...\}$ for each $k > 0$[BCST]. This is denoted FO($<$, mod) in the following strict containment diagram.



Two simple examples serve to illustrate the contrast between FO($<$, bit) and REG.

$$PARITY = \{w \in \{0, 1\}^* : w \text{ has an even number of ones}\}$$

$$MIDPOINT = \{0^n 1^n : n = 0, 1, 2...\}$$

A trivial finite automaton can recognize *PARITY*. Yet [FSS] show that *PARITY* cannot be first-order definable even with arbitrary numerical predicates. In contrast, a trivial FO($<$, bit) formula can express *MIDPOINT* (by using addition). However, *MIDPOINT* is the classic example of a nonregular language. For comparison purposes, note that

$$\{w \in \{0, 1\}^* : w \text{ has an equal number of zeros and ones}\}$$

is neither in REG nor FO($<$, bit), but is in ALOGTIME (since it can be checked by counting).

Comparison with Finite Automata

To help explain the discrepancies between regular languages and first-order logic, we turn to the table below which shows two differences between finite automata and the sequential deterministic model we propose.

	finite automata	*proposed model*
input access	single oblivious scan	multiple oblivious passes
flow of control	state machine	restricted state machine

Multiple Heads

Whereas a finite automata scans its input only once from left to right, our model allows for multiple heads, each of which is permitted to re-scan the input tape. To prevent positional information from being used as read/write storage, we restrict their movements (unidirectional with reset to the left edge) to be oblivious. This means that their locations depend only on the length of the input and not on its contents. We also include a mechanism whereby their relative and absolute positions can be queried.

Destructive Read

When a finite state machine is implemented, flip-flops are used to store the current state, while boolean gates combine this information with the current input bit to set the next state which is stored back in the same flip-flops. A fixed number of gates and binary storage elements assure a constant-space resource bound. In our model, gate types are restricted to AND, OR, NOT, and we insist that flip-flops are destructively read (making computations more akin to iterated boolean formulas), except when producing output (because it is never seen again).

Hardware Definition

We take a multi-head machine M consisting of a read-only input tape together with a fixed number of heads to scan the tape. It is equipped with head-crossing detectors which keep track of the relative left–right positions of any pair of heads, and (resettable) binary counters which keep track of the absolute position of each head. A mechanism is built-in whereby any particular bit of a head counter can be queried. There is also a write-only output tape whose head always moves forward whenever it writes an output bit.

While it is certainly possible to continue in this fashion and define our model in terms of time clocks and circuit diagrams, it is more convenient to describe our model in terms of a programming language, to make serial algorithms textually representable. By restricting storage to (read-once) boolean variables, the software will naturally constrain the model to a constant amount of (destructive-read) space. By limiting the constructs to composition and a strict form of definite loops (indexed by tape heads), oblivious flow of control and input/output access will be guaranteed.

Software Definition

The machine itself is controlled by a sequential program P, whose syntax assures that the machine obeys the polynomial-time and constant-space resource bounds,

and whose semantics assure the read-once restriction and oblivious head movement. We define these very simple programs by induction.

Booleans

The basic data type is a boolean, and only boolean variables are available for read-write storage. At any point in time, a *boolean variable* is either in the *read* or *unread* state. Apart from variables, there are other boolean values directly accessible in the machine model which do not have read restrictions. These are called *direct values*, which come from the input (using any tape head h as index), comparing the positions of any two tape heads, or querying any counter bit of a tape head. Sources for boolean values are summarized in this list:

`TRUE, FALSE`	boolean constants
b	a read-once/write boolean variable
`I[`h`]`	*false* if h is positioned over a zero on the tape, *true* otherwise
$i < j$	a comparison test which yields *true* iff head i is to the left of head j
`bit(`i`, `j`)`	*true* iff the jth column in the binary counter for head i is set[2]

Only the operations of `AND`, `OR`, `NOT` are allowed in combining these to form boolean expressions.

Assignments

If b is a boolean variable, and e is a boolean expression all of whose boolean variables are distinct and unread, then

$$b := e$$

assigns the value of e to b. Every boolean variable in e becomes read, and the status of b becomes unread (regardless of the state it was in before). Taken together, these conditions prevent a boolean variable from being read twice without an intervening write. Just note that this *read-once condition* can also be syntactically enforced in the further constructions below, even though we do not indicate precisely how it is done (essentially, keep track of which variables are required to be unread upon entering a block, and which are unread upon exiting a block).

Output

If e is any boolean expression, then

$$Out(e)$$

[2] Purists may object to this because the standard implementation of a binary counter requires more than $O(1)$ operations per "cycle," though the number of operations amortized over an entire loop is $O(1)$. In any event, this is similar to the read-only clock in Section 6 of [BI].

writes the value of e as the next bit on the output tape. However, the read/unread status of every boolean variable in e remains unchanged. Since the tape is write-only, there is no further access to this output, and hence none of the occurrences of the variables appearing in e are counted as read operations.

Composition

If P and Q are programs, then so is their sequential composition,

$$P;$$
$$Q$$

provided the read-once condition is not violated (by making sure that all variables that are required to be unread upon entry to Q are left unread upon exit from P).

Loops

Tape heads can serve as guarded parametrized controls for a loop. If the read-once condition is not violated, then the looping construct

$$\texttt{LOOP}\ h$$
$$P$$

binds tape head h to move over the input tape from the first cell to the last, performing one iteration of P for each such position $0, ..., n-1$, while moving h on the tape from the beginning (left) to the end (right). The position of h is not allowed to change inside P (assume for simplicity that loop heads are not re-bound, i.e., not nested with the same name). Also, assume that h returns to the left edge after completing the loop. Syntactic assurance of the read-once condition can be maintained by making sure all variables that are required to be unread upon entry to P are left unread upon exit from P.

Acceptance

By designating one of the boolean variables as the *result*, we can define string acceptance.

DEFINITION. Let P be a read-once constant-space sequential program as described above, and let $\bar{h}$ be a vector of head positions (range: $\{0, ..., |w|-1\}$). We say $\langle P, \bar{h}\rangle$ *accepts* $w \in \{0, 1\}^*$ if the result of running program P on input w with initial head positions starting at $\bar{h}$ is *true*. Omission of any or all of the positions $\bar{h}$ implies those heads begin at the left end of the tape (position zero). The corresponding language determined by P is

$$\{w \in \{0, 1\}^* : \langle P\rangle \text{ accepts } w\}.$$

3. EXAMPLES

We illustrate constant-space programs by two serial algorithms to provide both an example and a counterexample to the read-once condition.

Addition

The schoolbook algorithm for serial binary addition of two n-bit numbers, $a(n)\cdots a(1)+b(n)\cdots b(1)$, yields an n-bit sum $s(n)\cdots s(1)$ and a carry. For convenience, we use two input tapes both indexed by the same head, together with a single boolean variable c for the carry:

```
c := FALSE;                                   {initialize carry}
LOOP h                                          {from LSB to MSB}
  Out(a[h] XOR b[h] XOR c);                       {sum bit s(h)}
  c := a[h] AND b[h]  OR  c AND (a[h] OR b[h])
```

Notice how in the last line, the majority function has been carefully written with c factored so it only occurs once. Also, note that the occurrences of c used in producing the output are not counted as read operations (and XOR is used only as an abbreviation here).

Parity

In contrast, this simple program computes the parity of an input string $a(1)\cdots a(n)$.

```
p := FALSE;                                 {initialize parity}
LOOP h                                        {from LSB to MSB}
  p :=(a[h] AND NOT p) OR (p AND NOT a[h])          {XOR}
```

But here, in the last line, the boolean variable p violates the read-once condition because it must be read twice when forming the exclusive-or from the canonical base of boolean operations (although $a(h)$ may be read any number of times since it is a read-only input).

A consequence of our main theorem will be that the constant-space serial algorithm for addition implies the existence of a constant-time parallel algorithm, which I find quite surprising by itself, since the standard carry look-ahead algorithm was not a completely trivial observation in its time. Conversely, the existence of a constant-time parallel algorithm for binary addition implies the existence of a (read-once) constant-space serial algorithm. This phenomenon of time-space duality will be discussed further in Section 5.

4. MAIN RESULT

THEOREM. *A binary language $L \subseteq \{0,1\}^*$ is recognized by a read-once constant-space sequential program P if and only if it is definable by a first-order sentence ϕ over the class of binary string structures; i.e., $w \in L$ iff $\mathrm{A}_w \models \phi$.*

Note. In the interest of simplicity, we have chosen not to deal with output. However, it is a fairly easy extension of the theorem that the contents of the output tape is also governed by the same first-order behavior.

Proof. ($\Leftarrow$, the easy direction) We show by induction over the quantifier depth of a first-order $\{<, bit, U\}$-formula $\varphi(\bar{x})$ in prenex form that there is a program P such that

$$\mathrm{A}_w \models \varphi[\bar{h}] \Leftrightarrow \langle P, \bar{h} \rangle \text{ accepts } w,$$

where $\bar{h}$ is a tuple of numbers between 0 and $n-1$, whose length equals the length of $\bar{x}$.

Basis

If $\varphi(\bar{x})$ is quantifier-free, then it is easy to see that the value of the boolean sentence determined by $\varphi\langle \bar{h} \rangle$ can be computed by a loop-free program, since for all i and j in $\bar{h}$, the atomics $i < j$ and bit(i, j) are built-in direct values of the same name in the machine model, and because $U(h)$ can be directly read off the input by `I[h]`, since the tape head assigned to h is on that square by assumption.

Induction

Suppose $\varphi(\bar{x}) \equiv (\mathrm{Q}y)\, \psi(y, \bar{x})$, where $\mathrm{Q} \in \{\exists, \forall\}$. By induction hypothesis, $\psi[k, \bar{h}]$ is computed by a program $\langle P, (k, \bar{h}) \rangle$. To compute $\varphi[\bar{h}]$, we loop the first head around the program P and add an additional variable to store and compute the result of the quantification. Here is the program for existential quantification, whose result is b:

```
b := FALSE;
LOOP k
   P;                                   {with result "a"}
   b := b OR a;
```

Note that the program P must be repeatedly run for each position k. A similar dual program can be used for universal quantification.

($\Rightarrow$, the hard direction) Let V_P be the boolean variables of P left unread upon exit from P. The idea is to let initial head positions correspond to free variables in formulas and to express each boolean variable b in V_P by a first-order formula $\pi_b(\bar{x})$ such that the final result left in b after running P with initial head positions

$\bar{h}$ on input w is the same as the truth value of $\mathrm{A}_w \models \pi_b[\bar{h}]$. The proof proceeds by syntactic induction on P.

Assignments

In the base case, the program P is a single assignment statement `b := e`. Since e is just a combination of booleans, b can be represented quite easily by the single formula $\pi_b \equiv e'$, where e' is the formula which results from taking e and making the obvious substitutions.

Collections

To continue the argument, we will need a *collection* of first-order formulas, $\Pi = \{\pi_b(\bar{x}) : b \in V_P\}$ in order to express the values of all unread boolean variables upon exiting P. Boolean variables that are required to be unread upon entering P will appear as nullary atomic relations in these formulas and therefore, for fixed head positions, a program P can be thought of as a map from booleans to booleans. Furthermore, we will guarantee that each boolean variable occurs at most once in the collection Π and call this the *single occurrence property*. In the base case above, note that the collection Π is just the singleton $\{\pi_b\}$ and satisfies the single occurrence property since each boolean variable occurs at most once in e' in order for `b := e` to satisfy the read-once condition.

Compositions

The first inductive case is that of program composition `P; Q`. By induction hypothesis, there is a collection of first-order formulas $\Pi = \{\pi_b : b \in V_P\}$ expressing P, and a similar collection $\Theta = \{\theta_b : b \in V_Q\}$ expressing Q, each satisfying the single occurrence property. If a boolean variable a of Q does not appear in P, then adjoin to Π the identity formula $\pi_a \equiv a$. Similarly, if a boolean variable a of P does not appear in Q, then adjoin $\theta_a \equiv a$ to Θ. Note that neither of these changes affect the single occurrence property for Π or Θ.

Let $\theta_b[a \leftarrow \pi_a]$ denote for each boolean variable a in θ_b the substitution of the formula π_a. We claim that the *composition* $\Pi \circ \Theta = \{\theta_b[a \leftarrow \pi_a] : b \in V_P \cup V_Q\}$ expresses `P; Q` (Fig. 1).

Moreover, the single occurrence property for Θ guarantees that each formula π_a in Π is used exactly once to replace an occurrence of the boolean variable a in Θ. Since all the original boolean variables occurring in Θ are replaced in this manner, the only remaining boolean variables in $\Pi \circ \Theta$ are those occurring in Π. Since each

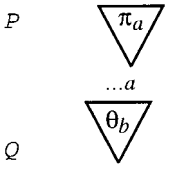


FIG. 1. Syntax tree for $\theta_b[a \leftarrow \pi_a]$.

formula π_a in Π was used exactly once, and since Π satisfies the single occurrence property, we can see that $\Pi \circ \Theta$ satisfies the single occurrence property too.

Ensembles

A further technicality is required to finish the proof. We associate with each program P an *ensemble* P, which is an eventually periodic function from the natural numbers to collections with the property that for each input size n, the collection $\mathsf{P}(n)$ represents P exactly as before. The finitely many collections in the range of P will be combined at the end of the proof to express the result of running P.

Going back to the base case, it is obvious that defining $\mathsf{P}(n) = \{\pi_b\}$ for all n creates an ensemble that works for assignments. To see that this added complication does not adversely affect the already completed inductive step for composition, just observe that given ensembles P and Q, their pointwise composition $\mathsf{P}(n) \circ Q(n)$ is eventually periodic and hence is clearly the desired ensemble.

Negation Normal Form

Before tackling loops, we first remove negations by rewriting P. Push all negations to the bottom, then replace every occurrence of `NOT b` by a new boolean variable b'. Now, follow each assignment `b := e` by `b' := NOT e`, where again we push negations to the bottom. Since every boolean variable appearing in e occurs once, both a boolean variable and its negation cannot both be in e, so the new program still satisfies the read-once condition.

Loops

The last inductive case is that of the loop program `LOOP h P`, which is by far the most difficult part of the argument. By induction hypothesis, we know that P is represented by an ensemble P. For each input size n, the collection $\Pi = P(n)$ has the property that for every $\pi_b(x) \in \Pi(x)$, the value of b after executing P is $\pi_b[h]$, for h equal to the position of the loop variable `h` (other unbound head positions have been omitted for clarity). The proof will be easier to understand if, at this point, the reader imagines n to be fixed (we will indicate later when it becomes necessary to vary n).

Syntactic Analysis

The single occurrence property insures that the *syntactic dependency graph* of Π, defined as the graph whose vertices are boolean variables in Π, and whose edges are given by $\{(a, b)\colon a \text{ appears in } \pi_b\}$ for all boolean variables a and b, has out-degree $\leqslant 1$. This means it looks like a bunch of disjoint whirlpools, each a single cycle (possibly trivial) which serves to connect the roots of several (possibly one) tree. See Fig. 2 for an example. Intuitively, this means that all programs essentially combine and shift values through a chain of variables.

Since boolean variables in disjoint components are independent of one another, then without loss of generality it suffices by syntactic decomposition to consider Π

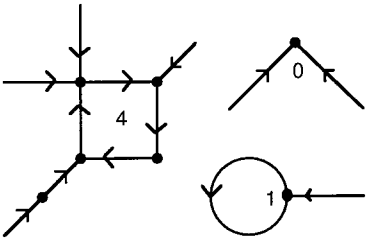


FIG. 2. As syntactic dependency graph.

to have only a single component. We now confine our discussion to such a case and define the *order* of Π to be the length m of the cycle that appears (see numbers in Fig. 2), reserving 0 for the case when there is a root instead of a cycle. Variables on the cycle are called *recursive*, and variables off the cycle are called *nonrecursive*. Define the *depth* of a nonrecursive variable to be its maximum distance from any leaf. Define the depth of Π to be the maximum distance d of the cycle (or root if there is no cycle) from any leaf.

Iterating the Loop

For input size n sufficiently large, the plan is to break up the n iterations of the loop into three sections. The first section, called the "leader," consists of an initial run of d iterations. This is followed by the "main section," which repeats an m-iterate block a total of $l=(n-d)\operatorname{div} m$ times. This in turn is followed by a "trailer" of the remaining $(n-d) \bmod m$ iterations. To accomplish this, we use the notation $\Pi(t)$ to be the result of substituting in every formula $\pi_b(x)$ in $\Pi(x)$ the (arithmetical) term t for every free occurrence of x (the variable representing the unbound head position h). We take advantage of the definability of arithmetic in FO($<$, bit) and freely use addition and multiplication for computations in the formulas we construct [L].

Leader

As long as $n \geqslant d$, the composition

$$\Lambda = \Pi(0) \circ \cdots \circ \Pi(d-1)$$

is clearly first-order expressible. Furthermore, since each nonrecursive boolean variable b has depth less than or equal to d, the corresponding $\lambda_b \in \Lambda$ will have no boolean variables occurring in it, and we call such formulas *explicit*. Since Λ is a finite power of Π under composition, the collection Λ satisfies the single occurrence property.

Cycle Contraction

We now proceed with the important task of contracting the cycle. If $m=0$, there is no cycle to contract, it is easy to see that the dependency graph is a tree, and

hence each boolean variable is nonrecursive. In this case, each boolean variable will have an explicit formula to describe its value, essentially consisting of the last d iterations of the loop. The details can be determined by continuing the proof for the case $m > 0$ and just skipping the parts which deal with a recursive boolean variable. On the other hand, if $m \geqslant 1$, then we simplify things by dividing the remaining $n-d$ iterations into $l = (n-d) \operatorname{div} m$ blocks of size m, writing each such block as the m-fold composition

$$\Theta(y) = \Pi(d + y \cdot m) \circ \cdots \circ \Pi(d + y \cdot m + m - 1) \qquad \text{where} \quad 0 \leqslant y < l.$$

Each increment of y corresponds to m iterations of the loop, and the dependency graph for Θ satisfies the single occurrence property (being a fixed power of Π). Furthermore, we have managed to obtain a dependency graph for Θ with m components of order 1, so that each recursive boolean variable is in its own cycle (Fig. 3).

Main Section

The main section consists of composing $\Theta(y)$, for each stage y, from $y = 0$ to $l-1$. Coming into the main section, all λ_b for b nonrecursive are explicit formulas (from the leader). We shall demonstrate that *all* of the partial iterates defined by

$$\Lambda \circ \Theta(0) \circ \Theta(1) \circ \cdots \circ \Theta(k-1) \qquad \text{for} \quad 0 \leqslant k \leqslant l$$

can be expressed by a *single* collection $\Psi(z)$, such that for $\psi_b(z) \in \Psi(z)$, $\psi_b[k]$ will express the value of b after $d + km$ iterations of the loop, and the final value of b after completing the main section will be given by $\psi_b(l)$. In other words, we are going to show that loop iteration (represented by the variable-length composition above) can be converted to quantification.

We first derive the formulas $\psi_b(z)$ for the values of nonrecursive boolean variables b at each stage of the main section, by induction on their depth in θ. Let $\theta_b(y)$ be the first-order formula for b in $\Theta(y)$. Supposing that ψ_a has been defined for a of smaller depth, define

$$\psi_b(z) \equiv (z = 0 \wedge \lambda_b) \vee (z > 0 \wedge \theta_b(z-1)[a \leftarrow \psi_a(z-1)]).$$

If b is a leaf in Θ, then θ_b contains no boolean variables, and the indicated substitution for a is vacuous. However, if b is not a leaf, then the replacements ψ_a

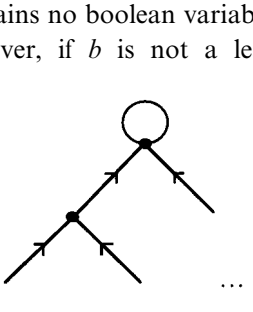
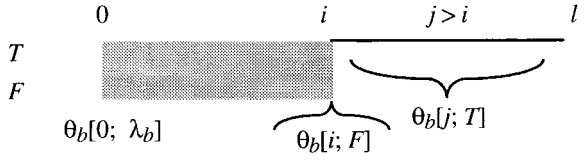


FIG. 3. One component of the syntactic dependency graph for Θ

have already been expressed by induction hypothesis (each a occurring in θ_b is by definition of smaller depth). Since λ_b contains no boolean variables, it follows by induction that neither does ψ_b, since as long as b is not recursive, neither is each a.

The formulas for the recursive variables are the most interesting part. Let b be a recursive boolean variable, and let $\theta_b(y) \in \Theta(y)$. The only occurrence of a recursive boolean variable in $\theta_b(y)$ is a single positive occurrence of b itself, by virtue of the fact that the dependency graph for Θ has order 1 and because we have eliminated negations while preserving the read-once condition. In particular, this makes $\theta_b(y)$ monotone in b.

We make the following fundamental observation: b is true at the end of a loop just in case b is true at some iteration and remains true for all subsequent iterations. In explaining this we'll use the notation T to stand for *true*, F for *false*, and $\theta_b[i; \tau]$ to stand for $\theta_b[i][b \leftarrow \tau]$, for any formula τ. If b is true at the end of the main section, then there must be some minimal stage i (possibly 0) in which b becomes (or is) true, i.e., $\theta_b[i; F]$ (or, $\theta_b[0; \lambda_b]$, where λ_b is the value of b upon entering the main section), and at all subsequent stages $j > i$, b remains true, i.e., $\theta_b[j; T]$. Conversely, if $\theta_b[i; F]$ for $i > 0$, then $\theta_b[i; T]$ holds by monotonicity, and hence b is true at stage i no matter what its previous value was (and if $\theta_b[0; \lambda_b]$ then b is true at stage $i = 0$). Furthermore, if $\theta_b[j; T]$ for $j > i$, then b remains true at all subsequent stages, and hence is true at the end of the main section. To help understand this, the following picture graphs the boolean value of b (shown on the vertical axis) through time (as shown by j on the horizontal axis) where it ends up true at the very end.



To simplify the formula we are about to construct, we combine all the uses of θ_b in the above figure. The cases $\theta_b[0; \lambda_b]$ for $i = 0$ and $\theta_b[i; F]$ for $i > 0$ can be merged into $\theta_b[i; i = 0 \wedge \lambda_b]$ for all $i \geqslant 0$. This can be cleverly merged with $\theta_b[j; T]$ for $j > i \geqslant 0$ to get $\theta_b[j; j > i \vee (i = 0 \wedge \lambda_b)]$ for $j \geqslant i \geqslant 0$. Introducing variables x, y, and z for i, j, and l resp. we obtain

$$\psi_b(z) \equiv (\exists x . 0 \leqslant x < z)(\forall y . x \leqslant y < z)\, \theta_b(y)[b \leftarrow y > x \vee (x = 0 \wedge \lambda_b); a \leftarrow \psi_a(y)],$$

where we substitute for the occurrences of all (nonrecursive) boolean variables a appearing in θ_b, their values upon entering the yth stage, which are given by the formulas $\psi_a(y)$ that we already determined above.

To see that the resulting collection Ψ satisfies the single occurrence property, note that for each nonrecursive boolean variable, ψ_a contains no boolean variables, as we observed earlier. For the remaining recursive boolean variables, recall that

the collection Λ satisfies the single occurrence property. Now observe that Ψ and $\Lambda \circ \Theta$ have the same dependency graph by examination of the constructed formulas. Since both Λ and Θ have the single occurrence property, their composition does also, which makes the induction go through.

Trailer

After completing $d + lm$ iterations of the loop as expressed by $\Psi(l)$, there are $r = (n - d) \bmod m$ remaining iterations of the loop, and these can be composed with Ψ to obtain

$$\Sigma = \Psi((n-d) \text{ div } m) \circ \Pi(n-r) \circ \cdots \circ \Pi(n-1)$$

which again has the single occurrence property because it is a finite composition. The astute reader should notice that if the syntactic structure of Π didn't change with n, the syntactic structure of Ψ wouldn't change with n. However, the syntactic structure of Σ does change with n because the number of compositions above ranges from 0 to $m-1$. Attempting to combine these m possibilities into a single collection would violate the single occurrence property. This is why we needed to use ensembles.

Wrapping Up

As n varies, we need to bring together all possible Σ into an ensemble S. To see that $\mathsf{S}(n) = \Sigma$ is eventually periodic, just observe that Σ only depends only on $\mathsf{P}(n) = \Pi$ which is eventually periodic, and $r = (n \bmod m) - (d \bmod m)$. To see that r is eventually periodic in n, note that $d \bmod m$ has the same periodicity as P since d and m both depend only on Π, and $n \bmod m$ is periodic because it repeats at the least common multiple of all the values that m ranges over. This completes the loop case and the induction.

Coda

To obtain a single sentence which expresses the final result of running a program P in which all heads are bound in loops, we need to stitch together the finitely many sentences $\pi_{result} \in \mathsf{P}(n)$ as n varies. Ignoring the enumeration of sentences for small $n \leqslant n_0$ equal to the length of the nonperiodic initial segment of P, let $\alpha_i = \pi_{result} \in \mathsf{P}(i + n_0)$ for $1 \leqslant i \leqslant p$ equal to the period of P. For $n > n_0$ the desired answer is then

$$\bigvee_{i=1}^{p} \alpha_i \wedge i = (n - n_0) \bmod p.$$

This is easily spliced with the finitely many cases for small n.

5. DIRECTIONS

Improvements

In the model we have defined, the use of a binary counter to monitor absolute head position is somewhat inelegant, and the looping construct to bind head movement is somewhat restrictive. It would be much nicer if the heads could be controlled by `MOVE` instructions, and loops replaced by `DO` h `TIMES` ... for some head position h. The lack of conditionals would retain obliviousness, and these instructions alone would be sufficient to obtain the necessary arithmetic, without having to keep track of relative and absolute head positions. Unfortunately, this may compute too much, and the success of this more aesthetic approach seems to depend on resolving a certain fundamental question in finite model theory; namely, does *iteration* (as a logical construct) over finite initial segments of the natural numbers with successor close at FO(+, *)? Surprisingly, iterated multiplication (exponentiation, ^) is in FO(+, *). However, a more promising approach which doesn't rely on complexity theoretic separations might be to just devise some simple mechanism of tying head movements to a global system clock using, for example, frequency dividers (cascaded chains of flip-flops).

Extensions to Other Complexity Classes

It is relatively easy to obtain a sequential deterministic characterization of ALOGTIME (uniform-NC^1) by dropping the destructive read restriction in our constant-space model (in fact the proof would be much easier, cf. [BI]). This identifies ALOGTIME in a reasonably elegant manner with constant-space serial algorithms. To my mind the real challenge is to find a similar sequential deterministic model for uniform constant-depth threshold circuits (TC^0). Presumably, the model might use integer variables with the read-once restriction. It would also be instructive to directly prove the containment $TC^0 \subseteq NC^1$ in this setting, demonstrating a concrete constant-space serial algorithm for counting.

Time/Space Duality

We observed in Section 3 that the standard schoolbook algorithm for binary addition is a read-once constant-space serial algorithm. If you go through the proof of our main theorem, you will obtain a first-order formula for it which can be seen to be virtually the same as the classic carry look-ahead method:

$$s(i) = a(i) \oplus b(i) \oplus c(i), \qquad \text{where}$$
$$c(i) = (\exists j)[\, j < i \wedge a(i) \wedge b(i) \wedge (\forall k)[\, j < k < i \rightarrow (a(i) \vee b(i))\,]\,].$$

Perhaps the best-known (and probably oldest) parallel algorithm, it says that there is a carry into a column if and only if some previous column generated one and every subsequent column propagated it.

A somewhat surprising consequence of our work is that in general, for polynomial bounds on size and length, destructive read constant-space serial algorithms

are equivalent to constant-time parallel algorithms. Classical parallel-time/serial-space duality is a phenomenon that appears only to extend down to resource bounds of $O(\log n)$ [H]. However, the seemingly peculiar read-once restriction has allowed us to take this duality all the way down to $O(1)$ time or space. By modifying our model to allow dynamically allocated new storage (additional boolean variables) it should be possible to use Immerman's iterated first-order formulas to extend our results and provide a tight correspondence between read-once space and quantifier-depth. One particularly intriguing possibility is a duality theorem for TC^0, involving the as yet undiscovered sequential model mentioned above.

Multiplication

Since n-bit multiplication is in TC^0, it is instructive to consider the usual schoolbook algorithm for binary multiplication as an example of a serial algorithm with integer variables.

$$\begin{array}{r} d_{n-1} \ldots d_0 \\ \times\, e_{n-1} \ldots e_0 \\ \hline = p_{2n-1} \ldots\ldots p_0 \end{array}$$

The partial products (not pre-computed) are added right to left in columns, and the partial sums for each column accumulated top to bottom, with a multi-bit carry into the next column.

```
s := 0;                                  {initialize partial sum}
LOOP i FROM 0 TO 2*n-1                       {go from LSB to MSB}
  LOOP j FROM 0 TO i                          {sum for ith column}
    s := s + d(j) * e(i-j);                 {add next term in sum}
  p(i) := s mod 2;                           {output product bit}
  s := s div 2                              {carry to next column}
```

Notice that this uses variable loop bounds and the read/write integer variable s of $O(\log n)$ bits. It is intriguing to wonder if a parallel dual to this serial algorithm might not provide a simplified witness to the fact that multiplication can be performed in uniform-TC^0. Unlike carry look-ahead, the current parallel constructions are rather more complex and involved than this simple and intuitive serial method [IL].

ACKNOWLEDGEMENTS

I am grateful for discussions with Dave Mix Barrington and Sorin Istrail, who introduced me to the equivalence between constant-depth circuits and read-once constant-width branching programs [IZ]. I also appreciate discussions with Sam Buss, Dejan Zivkovic, Steve Tate, Eric Allender, Howard Straubing, Ken Regan, Scott Weinstein, Eric Rosen, Peter Clote, and Ian Parberry (who made the particularly good suggestion of using tape heads instead of the random-access method I was using). In

revising this paper, I thank the anonymous referees for many constructive suggestions and the editor Daniel Leivant for his encouragement and patience. A special note of appreciation goes to my wife Suzanne for her support and assistance in typing and completing the revisions.

Final manuscript received December 23, 1997

REFERENCES

[ABI] Allender, E., Balcázar, and Immerman, N. (1997), A first-order isomorphism theorem, *SIAM J. Comput.* **26**(2), 557–567.

[AG] Allender, E., and Gore, V. (1991), Rudimentary reductions revisited, *Inform. Process. Lett.* **40**, 89–95.

[B] Buss, S. (1993), Algorithms for Boolean formula evaluation and for tree contraction, *in* "Arithmetic, Proof Theory, and Computational Complexity" (Peter Clote and Jan Krajíček, Eds.), pp. 95–115, Oxford Univ. Press, London.

[BCST] Mix Barrington, D., Compton, K., Straubing, H., and Thérien, D. (1992), Regular languages in NC^1, *JCSS*, 478–499.

[BI] Mix Barrington, D., and Immerman, N. (1997), Time, hardware, and uniformity, *in* "Complexity Theory Retrospective II" (L. A. Hemaspaandra and A. L. Selman, Eds.), Springer-Verlag, Berlin.

[BIS] Mix Barrington, D., Immerman, N., and Straubing, H. (1990), On uniformity in NC^1, *JCSS* **41**, 274–306.

[C] Clote, P. (1990), Sequential, machine-independent characterizations of the parallel complexity classes AlogTIME, AC^k, NC^k and *NC*, *in* "Feasible Mathematics" (S. Buss and P. Scott, Eds.), Birkhäuser, Basel.

[D] Dahlhaus, E. (1984), "Reduction to NP-complete Problems by Interpretations," LNCS 171, pp. 357–365, Springer-Verlag, Berlin.

[D′] Dawar, A. (1995), Generalized quantifiers and logical reducibilities, *J. Logic Comput.* **5**(2), 213–226.

[DDLW] Dawar, A., Doets, K., Lindell, S., and Weinstein, S., "Elementary Properties of the Finite Ranks," Technical Report MS-CIS-96-24 of the Department of Computer and Information Science, University of Pennsylvania. [To appear in Math. Log. Quar.]

[E] Enderton, H. (1972), "A Mathematical Introduction to Logic," Academic Press, San Diego.

[FSS] Furst, M., Saxe, J. B., and Sipser, M. (1984), Parity, circuits, and the polynomial-time hierarchy, *Math. Syst. Theory* **17**, 13–27.

[G] Gurevich, Y. (1988), Logic and the challenge of computer science, *in* "Trends in Theoretical Computer Science" (Egon Börger, Ed.), pp. 1–57, Computer Science Press.

[H] Hong, J. W. (1986), "Computation: Computability, Similarity, and Duality," Wiley, New York.

[I] Immerman, N. (1989), Expressibility and parallel complexity, *SIAM J. Comput.* **18**(3), 625–638.

[IL] Immerman, N., and Landau, S. (1995), The complexity of iterated multiplication, *Information Comput.* **116**(1), 103–116.

[IZ] Istrail, S., and Zivkovic, D. (1994), Bounded-width polynomial-size Boolean formulas compute exactly those functions in AC^0, *Information Process. Lett.* **50**, 211–216.

[L] Lindell, S. (1992), A purely logical characterization of circuit uniformity, *IEEE Struct. Complexity Theory*, 185–192.

[S] Straubing, H. (1994), "Finite Automata, Formal Logic, and Circuit Complexity," Birkhäuser, Basel.