EXPRESSIBILITY AND PARALLEL COMPLEXITY

NEIL IMMERMAN[*]

SIAM J. of Comput. 18 (1989), 625-638.

Abstract

It is shown that the time needed by a concurrent-read, concurrent-write parallel random access machine (CRAM) to check if an input has a certain property is the same as the minimal depth of a first-order inductive definition of the property. This in turn is equal to the number of "iterations" of a first-order sentence needed to express the property.

The second contribution of this paper is the introduction of a purely syntactic uniformity notion for circuits. It is shown that an equivalent definition for the uniform circuit classes AC^i, $i \geq 1$ is given by first-order sentences "iterated" $\log^i n$ times. Similarly, uniform AC^0 is defined to be the first-order expressible properties (which in turn is equal to constant time on a CRAM by our main theorem).

A corollary of our main result is a new characterization of the Polynomial-Time Hierarchy (PH): PH is equal to the set of languages accepted by a CRAM using exponentially many processors and constant time.

Key words. Computational complexity, parallel complexity, first-order expressibility, polynomial-time hierarchy

AMS(MOS) subject classification. 68025

1 Introduction

Parallel time on a random access machine has a surprisingly simple mathematical definition involving well-studied objects of mathematical logic. We

[*]This research was supported in part by the Mathematical Sciences Research Institute, Berkeley, California, and by National Science Foundation grants DCR-8603346 and CCR-8806308.

show that the time needed by a concurrent-read, concurrent-write parallel random access machine (CRAM) to check if an input has a certain property is the same as the minimal depth of a first-order inductive definition of the property. This in turn is equal to the number of "iterations" of a first-order sentence needed to express the property.

We now state our main result. (See §2 for relevant definitions. In particular, the iteration of a first-order sentence is defined in §2.2, and the CRAM is defined in §2.3. The definition of the CRAM differs from the standard definition of the CRCW PRAM in [17] only in that a processor may shift a word of local memory by any polynomial number of bits in unit time. It follows from our results that for parallel time greater than or equal to $\log n$ there is no distinction between the models with and without the Shift instruction.)

Theorem 1.1 *Let S be a set of structures of some vocabulary τ. For example, S is a set of boolean strings, or a set of graphs, etc. For all polynomially bounded, parallel time constructible $t(n)$, the following are equivalent:*

1. *S is recognizable by a* CRAM *in parallel time $t(n)$, using polynomially many processors.*
2. *There exists a first-order sentence φ such that the property S for structures of size at most n is expressed by φ iterated $t(n)$ times.*
3. *S is definable as a uniform first-order induction whose depth, for structures of size n, is at most $t(n)$.*

For $t(n) \geq \log n$, the equivalence of (1) and (2) in Theorem 1.1 may also be obtained by combining a result of Ruzzo and Tompa relating CRAMs to alternating Turing machines [17, Thm. 3], together with a result of ours relating alternating Turing machines to first-order expressibility [9, Thm. B.4]. In order to prove the theorem for $t(n) < \log n$, we were forced to modify the models slightly, adding the Shift operation to the CRAMs and adding BIT as a new logical relation to our first-order language (see §2). We believe that the naturalness of Theorem 1.1 justifies these modifications.

This paper is organized as follows. In §2 we give all relevant definitions. In §3 we prove our main result. In §4 we give a more detailed analysis of the bounds in Theorem 1.1. We show that the number of distinct variables in a first-order inductive definition is closely tied to the number of processors in the corresponding CRAM.

Until now, a principal unaesthetic feature of the theory of complexity via boolean circuits was that one had resorted to Turing machines to define the uniformity conditions for circuits [15]. As a corollary to Theorem 1.1, we obtain a purely syntactic uniformity notion for circuits. In §5 we describe this result as well as other relations between circuits and first-order complexity.

As another corollary to Theorem 1.1, we present in §6 a new characterization of the Polynomial-Time Hierarchy (PH): PH is equal to the set of languages recognized by a CRAM using exponentially many processors and constant time. In §7 we give some suggestions for future work in this area.

2 Background and definitions

2.1 First-order logic

We begin this section by making some precise definitions concerning first-order logic. For more information see [4].

A *vocabulary* $\tau = \langle \underline{R}_1^{a_1}, \cdots, \underline{R}_k^{a_k}, \underline{c}_1, \cdots, \underline{c}_r \rangle$ is a tuple of relation symbols and constant symbols. $\underline{R}_i^{a_i}$ is a relation symbol of arity a_i. In the sequel we will usually omit the superscripts and the underlines to improve readability. A finite *structure* of vocabulary τ is a tuple, $\mathcal{A} = \langle \{0, 1, \cdots, n-1\}, R_1^{\mathcal{A}}, \cdots, R_k^{\mathcal{A}}, c_1^{\mathcal{A}}, \cdots, c_r^{\mathcal{A}} \rangle$, consisting of a universe $|\mathcal{A}| = n = \{0, \cdots, n-1\}$ and relations $R_1^{\mathcal{A}}, \cdots, R_k^{\mathcal{A}}$ of arities $a_1, \cdots, a_k$ on $|\mathcal{A}|$ corresponding to the relation symbols $\underline{R}_1^{a_1}, \cdots, \underline{R}_k^{a_k}$ of τ, and constants $c_1^{\mathcal{A}}, \cdots, c_r^{\mathcal{A}}$ from $|\mathcal{A}|$ corresponding to the constant symbols $\underline{c}_1, \cdots, \underline{c}_r$ from τ.

For example, a graph on n vertices, $G = \langle \{0...n-1\}, E \rangle$, is a structure whose vocabulary $\tau_0 = \langle \underline{E}^2 \rangle$ has a single binary relation symbol. Similarly, a binary string of length n is a structure $S = \langle \{0...n-1\}, M \rangle$ whose vocabulary $\tau_1 = \langle \underline{M}^1 \rangle$ consists of a single unary relation symbol. Here the ith bit of S is 1 if and only if $S \models M(i)$.

Let the symbol "$\leq$" denote the usual ordering on the natural numbers. We will include $\leq$ as a logical relation in our first-order languages. This seems necessary in order to simulate machines whose inputs are structures given in some order. It is convenient to include logical constant symbols, $0, 1, \cdots$, referring to the zeroth, first, etc., elements of the universe, respectively. (If the universe is smaller than a given constant, then interpret that constant as 0.) We also include the logical predicate BIT, where $\text{BIT}(x, y)$ holds if and only if the xth bit in the binary expansion of y is a one.[1]

[1]The relation BIT is crucial for the truth of Theorem 1.1, when $t(n) < \log n$, and for the plausibility of Definition 5.5.

We now define the *first-order language* $\mathcal{L}(\tau)$ to be the set of formulas built up from the relation and constant symbols of τ and the logical relation and constant symbols, $=, \leq, \mathrm{BIT}, 0, 1, \cdots$, using logical connectives, $\wedge, \vee, \neg$, variables, $x, y, z, \cdots$, and quantifiers, $\forall, \exists$.

We will think of a *problem* as a set of structures of some vocabulary τ. It suffices to consider only problems on binary strings, but it is more interesting to be able to talk about other vocabularies, e.g., graph problems, as well. For definiteness, we will fix a scheme for coding an input structure as a binary string. If $\mathcal{A} = \langle \{0, 1, \cdots, n-1\}, R_1^{\mathcal{A}} ... R_k^{\mathcal{A}}, c_1^{\mathcal{A}} ... c_r^{\mathcal{A}} \rangle$, is a structure of type τ, then $\mathcal{A}$ will be encoded as a binary string $\mathrm{bin}(\mathcal{A})$ of length $I(n) = n^{a_1} + \cdots + n^{a_k} + r\lceil \log n \rceil$, consisting of one bit for each a_i-tuple, potentially in the relation R_i, and $\lceil \log n \rceil$ bits to name each constant, c_j. Thus we reserve n to indicate the size of the universe of the input structure. $I(n)$, the length of $\mathrm{bin}(\mathcal{A})$, is polynomially related to n, and in the case where τ consists of a single unary relation – i.e. inputs are binary strings – $I(n) = n$.

Define the complexity class FO to be the set of all first-order expressible problems. We will see in §5 that FO is a uniform version of the circuit class AC^0. (See also [1] where it is shown that FO is equal to deterministic log time uniform AC^0.)

Example 2.1 An example of a first-order expressible property is addition.[2] In order to turn addition into a yes/no question, we can let our input have the vocabulary $\tau_a = \langle A, B, k \rangle$ consisting of two unary relations and a constant symbol. In a structure $\mathcal{A}$ of vocabulary τ_a, the relations A and B are binary strings of length $n = |\mathcal{A}|$. We will say that $\mathcal{A}$ satisfies the addition property if the kth bit of the sum of A and B is one.

In order to express addition, we will first express the carry bit,

$$\mathrm{CARRY}(x) \equiv (\exists y < x)[A(y) \wedge B(y) \wedge (\forall z . y < z < x) A(z) \vee B(z)]$$

Then with $\oplus$ standing for exclusive or, we can express PLUS,

$$\mathrm{PLUS}(x) \;\equiv\; A(x) \oplus B(x) \oplus \mathrm{CARRY}(x)$$

Thus the sentence expressing the addition property is $\mathrm{PLUS}(k)$. $\square$

[2]This is a standard construction, see e.g., [17].

2.2 Iterating first-order sentences

To describe properties that are not in AC^0, we need languages that are more expressive than FO. We now recall the definition of the complexity classes $\mathrm{FO}[t(n)]$[3]. Intuitively, $\mathrm{FO}[t(n)]$ consists of those problems that may be described by a first-order sentence "iterated $t(n)$ times."

Let x be a variable and M a quantifier free formula. We will use the notation $(\forall x.M)\psi$ – read, "for all x such that M, ψ," – to abbreviate $(\forall x)(M \to \psi)$. Similarly we will write $(\exists x.M)\psi$ – read, "there exists an x such that M, ψ," – to abbreviate $(\exists x)(M \wedge \psi)$. We will call the expressions $(\forall x.M)$ and $(\exists x.M)$ *restricted quantifiers.* Let a *quantifier block* be a finite sequence of restricted quantifiers: $\mathrm{QB} = (Q_1 x_1.M_1)\cdots(Q_k x_k.M_k)$. We will use the notation $[\mathrm{QB}]^t$ to denote the quantifier block QB repeated t times. I mean this literally:

$$[\mathrm{QB}]^t \;=\; \underbrace{\mathrm{QB}\,\mathrm{QB}\,\mathrm{QB}\cdots\mathrm{QB}}_{t \text{ times}} \;.$$

Note that for any quantifier-free formulas $M_0, M_1, \cdots, M_k \in \mathcal{L}(\tau)$, and any $i \in \mathbf{N}$, the expression $[\mathrm{QB}]^i M_0$ is a well-formed formula in $\mathcal{L}(\tau)$.

Definition 2.2 Let $t : \mathbf{N} \to \mathbf{N}$ be any function, and let τ be any vocabulary. A set C of structures of vocabulary τ is a member of $FO[t(n)]$ if and only if there exists a quantifier block QB and a quantifier-free formula M_0 from $\mathcal{L}(\tau)$, such that if we let $\varphi_n = [\mathrm{QB}]^{t(n)} M_0$, for $n = 1, 2, \cdots$, then for all structures G of vocabulary τ with $|G| = n$,

$$G \in C \;\Leftrightarrow\; G \models \varphi_n \;.$$

A more traditional way to iterate formulas is by making inductive definitions, [14], [10]. Let IND-DEPTH$[t(n)]$ be the set of problems expressible as a uniform induction that requires depth of recursion at most $t(n)$ for structures of size n. In [7], Harel and Kozen introduce a programming language called IND, which is closely tied to inductive definitions. They prove that the execution time for their IND programs is equal to the depth of the inductive definitions that describe the programs' input output behavior. Let IND-TIME$[t(n)]$ be the set of languages accepted by IND programs using $O[t(n)]$ steps for inputs of size n. Then:

[3]The notation $\mathrm{FO}[t(n)]$ was first used in [12]; however, the same classes were defined in [10] using the notation $\mathrm{IQ}[t(n)]$, standing for "iterated queries." See [13] for a survey of descriptive complexity.

Fact 2.3 ([7]) *For all* $t(n)$,

$$\text{IND-TIME}[t(n)] \;=\; \text{IND-DEPTH}[t(n)]\ .$$

This fact, together with Theorem 1.1, shows that there is a simple, high level programming language for which time corresponds exactly to time on a CRAM. In the remainder of this paper we write IND$[t(n)]$ to signify IND-TIME$[t(n)]$ as well as IND-DEPTH$[t(n)]$.

The following fact relates IND$[t(n)]$ to FO$[t(n)]$. This fact follows easily from Moschovakis' Canonical Form for Positive Formulas, [14].

Fact 2.4 ([10], [14]) *For all* $t(n)$,

$$\text{IND}[t(n)] \subseteq \text{FO}[t(n)]\ .$$

(In particular, a property in IND$[t(n)]$ *is expressible as a* FO$[t(n)]$ *property in which* $M_0 \equiv$ *false, cf. Definition* 2.2.*)*

Example 2.5 We show how to transfer a $\log n$ depth inductive definition of the transitive closure of a graph to an equivalent FO$[\log n]$ definition.

Let E be the edge predicate for a graph G with n vertices. We can inductively define E^*, the reflexive, transitive closure of G, as follows:

$$E^*(x,y) \;\equiv\; x = y \vee E(x,y) \vee (\exists z)(E^*(x,z) \wedge E^*(z,y))\ .$$

Let $P_n(x,y)$ mean that there is a path of length at most n from x to y. Then we can rewrite the above definition of E^* as:

$$P_n(x,y) \;\equiv\; x = y \vee E(x,y) \vee (\exists z)(P_{n/2}(x,z) \wedge P_{n/2}(z,y))\ .$$

This can be rewritten:

$$P_n(x,y) \;\equiv\; (\forall z.M_1)(\exists z)(P_{n/2}(x,z) \wedge P_{n/2}(z,y))\ ,$$

where $M_1 \equiv \neg(x = y \vee E(x,y))$. Note that there is no free occurrence of the variable z after the $\forall z$ quantifier. Thus, in this case $(\forall z.M_1)\alpha$ is equivalent to $(M_1 \rightarrow \alpha)$. Next,

$$P_n(x,y) \;\equiv\; (\forall z.M_1)(\exists z)(\forall uv.M_2)(P_{n/2}(u,v))\ ,$$

where $M_2 \equiv (u = x \wedge v = z) \vee (u = z \wedge v = y)$. Now,

$$P_n(x,y) \;\equiv\; (\forall z.M_1)(\exists z)(\forall uv.M_2)(\forall xy.M_3)(P_{n/2}(x,y))\ ,$$

where $M_3 \equiv (x = u \wedge y = v)$. Thus,

$$P_n(x,y) \equiv [\mathrm{QB}]^{\lceil \log n \rceil}(P_1(x,y)) ,$$

where $\mathrm{QB} = (\forall z.M_1)(\exists z)(\forall uv.M_2)(\forall xy.M_3)$. Note that

$$P_1(x,y) \equiv [\mathrm{QB}](\mathit{false}) .$$

It follows that

$$P_n(x,y) \equiv [\mathrm{QB}]^{\lceil 1+\log n \rceil}(\mathit{false}) ,$$

and thus $E^* \in \mathrm{FO}[\log n]$ as claimed. $\square$

2.3 Concurrent random access machines

We define the concurrent random access machine (CRAM) to be essentially the concurrent read, concurrent write parallel random access machine (CRCW PRAM) described in [17]. A CRAM is a synchronous parallel machine such that any number of processors may read or write into any word of global memory at any step. If several processors try to write into the same word at the same time, then the lowest-numbered processor succeeds.[4] In addition to assignments, the CRAM instruction set includes addition, subtraction, and branch on less than. Each processor also has a local register containing its processor number.

The difference between the CRAM and the CRCW PRAM described in [17] is that we also include a Shift instruction. Shift(x, y) causes the word x to be shifted y bits to the right. Without Shift, CRAM$[t(n)]$ would be too weak to simulate FO$[t(n)]$ for $t(n) < \log n$. The reason behind the Shift operation for CRAMs and the corresponding BIT predicate for first-order logic is that each bit of global memory should be available to every processor in constant time.

Let CRAM$[t(n)]$ be the set of problems accepted by a CRAM using polynomially many processors and time $O[t(n)]$. Recall that we encode an input structure $\mathcal{A} = \langle\{0,1,\cdots,n-1\}, R_1^{\mathcal{A}},\cdots,R_k^{\mathcal{A}}, c_1^{\mathcal{A}},\cdots,c_r^{\mathcal{A}}\rangle$, as the binary string bin$(\mathcal{A})$ of length $I(n) = n^{a_1} + \cdots + n^{a_k} + r\lceil \log n \rceil$, Where a_i is the arity of the ith input relation. The input string is placed one bit at a time in the first $I(n)$ global memory locations.[5]

[4] This is the "priority write" model. Our results remain true if instead we use the "common write" model, in which the program guarantees that different values will never be written to the same location at the same time. See Corollary 3.4.

[5] We show in Corollary 3.4 that if placement of the input is varied, e.g., if the first

3 Proof of the main theorem

Theorem 1.1 follows immediately from three containments: Fact 2.4, and the following two lemmas.

Lemma 3.1 *For any polynomially bounded $t(n)$ we have,*

$$\mathrm{CRAM}[t(n)] \subseteq \mathrm{IND}[t(n)] .$$

Proof We want to simulate the computation of a CRAM M. On input $\mathcal{A}$, a structure of size n, M runs in $t(n)$ synchronous steps, using $p(n)$ processors, for some polynomial $p(n)$. Since the number of processors, the time, and the memory word size are all polynomially bounded, we need only a constant number of variables $x_1, \cdots, x_k$, each ranging over the n element universe of $\mathcal{A}$, to name any bit in any register belonging to any processor at any step of the computation. We can thus define the contents of all the relevant registers for any processor of M, by induction on the time step.

We now specify the CRAM model more precisely. We may assume that each processor has a finite set of registers including the following: Processor, containing the number between 1 and $p(n)$ of the processor; Address, containing an address of global memory; Contents, containing a word to be written into or read from global memory; and Program_Counter, containing the line number of the instruction to be executed next. The instructions to be simulated are limited to the following:

- READ: Read the word of global memory specified by Address into Contents.
- WRITE: Write the Contents register into the global memory location specified by Address.
- OP $R_a\, R_b$: Perform OP on R_a and R_b, leaving the result in R_b. Here OP may be Add, Subtract, or Shift.
- MOVE $R_a\, R_b$: Move R_a to R_b.
- BLT $R\, L$: Branch to line L if the contents of R is less than zero.

$I(n)/\log n$ words of memory contain $\log n$ bits each of the input, or even if all $I(n)$ bits are placed in the first word, then all our results remain unchanged. Note that this is not true of the models used in [2], for example. There processors are assumed to have unlimited power and thus the partition of the inputs is crucial.

It is straightforward to write a first-order inductive definition for the relation VALUE$(\overline{p}, \overline{t}, \overline{x}, r, b)$ meaning that bit $\overline{x}$ in register r of processor $\overline{p}$ just after step $\overline{t}$ is equal to b. Note that since the number of processors, the time, and the word size are all polynomially bounded, a constant number of variables ranging from 0 to $n-1$ suffice to specify each of these values.

The inductive definition of the relation VALUE$(\overline{p}, \overline{t}, \overline{x}, r, b)$ is a disjunction depending on the value of $\overline{p}$'s program counter at time $\overline{t}-1$. The most interesting case is when the instruction to be executed is READ. Here we simply find the most recent time $\overline{t'} < \overline{t}$ at which the word specified by $\overline{p}$'s Address register at time $\overline{t}$ was written into, and the lowest numbered processor $\overline{p'}$ that wrote into this address at time $\overline{t'}$. In this way we can access the answer, namely the $\overline{x}$th bit of $\overline{p'}$s Contents register at time $\overline{t'}$.

It remains to check that Addition, Subtraction, BLT, and Shift are first-order expressible, and that we can express the fact that each processor begins with its own processor number in its Processor register. Addition was done in Example 2.1. In a similar way we can express Subtraction, and Less Than. The main place we need the BIT relation is to express the fact that the initial contents of each processor's Processor register is its processor number. The relation BIT allows us to translate between variable numbers and words in memory. Using BIT we can also express addition on variable numbers and thus express the Shift operation.

Thus we have described an inductive definition of the relation VALUE, coding M's entire computation. Furthermore, one iteration of the definition occurs for each step of M. $\square$

Lemma 3.2 *For polynomially bounded, and parallel time constructible* $t(n)$,

$$\mathrm{FO}[t(n)] \ \subseteq\ \mathrm{CRAM}[t(n)]\ .$$

Proof Let the FO$[t(n)]$ problem be determined by the following quantifier-free formulas and quantifier block,

$$M_0, M_1, \cdots, M_k, \quad \mathrm{QB} = (Q_1 x_1 . M_1) \cdots (Q_k x_k . M_k)\ .$$

Our CRAM must test whether an input structure $\mathcal{A}$ satisfies the sentence,

$$\varphi_n \ \equiv\ [\mathrm{QB}]^{t(n)} M_0\ .$$

The CRAM will use n^k processors and n^{k-1} bits of global memory. Note that each processor has a number $a_1 \cdots a_k$ with $0 \le a_i < n$. Using the Shift operation it can retrieve each of the a_is in constant time.[6]

The CRAM will evaluate φ_n from right to left, simultaneously for all values of the variables $x_1, \cdots, x_k$. For $0 \le r \le t(n) \cdot k$, let,

$$\varphi_n^r \;\equiv\; (Q_i x_i.M_i) \cdots (Q_k x_k.M_k)[\text{QB}]^q M_0\;,$$

where $r = k \cdot (q+1) + 1 - i$. Let $x_1 \cdots \hat{x}_i \cdots x_k$ be the $k-1$-tuple resulting from $x_1 \cdots x_k$ by removing x_i. We will now give a program for the CRAM which is broken into rounds, each consisting of three processor steps such that:

$(*)$ Just after the rth round, the contents of memory location $a_1 \cdots \hat{a}_i \cdots a_k$ is 1 or 0 according to whether $\mathcal{A} \models \varphi_n^r(a_1, \cdots, a_k)$.

Note that x_i is not free in φ_n^r! At the rth round, processor number $a_1 \cdots a_k$ executes the following three instructions according to whether $Q_i = \exists$ or $Q_i = \forall$:

```
{$Q_i = \exists$}
        1. $b \leftarrow \text{loc}(a_1 \cdots \hat{a}_{i+1} \cdots a_k)$;
        2. $\text{loc}(a_1 \cdots \hat{a}_i \cdots a_k) \leftarrow 0$;
        3. if $M_i(a_1, \cdots, a_k)$ and $b$ then $\text{loc}(a_1 \cdots \hat{a}_i \cdots a_k) \leftarrow 1$;
{$Q_i = \forall$}
        1. $b \leftarrow \text{loc}(a_1 \cdots \hat{a}_{i+1} \cdots a_k)$;
        2. $\text{loc}(a_1 \cdots \hat{a}_i \cdots a_k) \leftarrow 1$;
        3. if $M_i(a_1, \cdots, a_k)$ and $\neg b$ then $\text{loc}(a_1 \cdots \hat{a}_i \cdots a_k) \leftarrow 0$;
```

It is not hard to prove by induction that $(*)$ holds, and thus that the CRAM simulates the formula. $\square$

Remark 3.3 The proof of Lemma 3.2 provides a very simple network for simulating a FO$[t(n)]$ property. The network has n^{k-1} bits of global memory and kn^k gates, where k is the number of distinct variables in the quantifier

[6]This is obvious if n is a power of 2. If not, we can just let each processor break its processor number into k $\lceil \log n \rceil$-tuples of bits. If any of these is greater than or equal to n, then the processor should do nothing during the entire computation.

block. Each gate of the network is connected to two bits of global memory in a simple connection pattern. The blowup of processors going from CRAM to FO to CRAM seems large (cf. Corollary 4.1); however, it is plausible to build first-order networks with billions of processing elements, i.e. gates, thus accommodating fairly large n and moderately large k.

An immediate corollary of Theorem 1.1 is that the complexity class CRAM$[t(n)]$ is not affected by minor changes in how the input is arranged, nor in the global memory word size, nor even by a change in the convention on how write conflicts are resolved.

Corollary 3.4 *For any function $t(n)$, the complexity class* CRAM$[t(n)]$ *is not changed if we modify the definition of a* CRAM *in any consistent combination of the following ways. (By consistent we mean that we don't allow input words larger than the global word size, nor larger than the allowable length of applications of Shift.)*

1. *Change the input distribution so that either* (a) *the entire input is placed in the first word of global memory, or* (b) *the $I(n)$ bits of input are placed* $\log n$ *bits at a time in the first $I(n)/\log n$ words of global memory.*
2. *Change the global memory word size so that either* (a) *The global word size is one, i.e. words are single bits. (Local registers do not have this restriction so that the processor's number may be stored and manipulated.), or* (b) *The global word size is bounded by $O[\log n]$.*
3. *Modify the Shift operation so that shifts are limited to the maximum of the input word size and of the log base 2 of the number of processors.*
4. *Remove the polynomial bound on the number of memory locations, thus allowing an unbounded global memory.*
5. *Instead of the priority rule for the resolution of write conflicts, adopt the common write rule in which different processors never write different values into the same memory location at a given time step.*

Proof The proof is that Lemmas 3.1 and 3.2 still hold with any consistent set of these modifications. This is immediate for Lemma 3.1. For Lemma 3.2, we must only show that processor number $a_1 \cdots a_k$ still has the power in constant time to evaluate the quantifier-free formula $M_i(a_1, \cdots, a_k)$, and to name the global memory location $a_1 \cdots \hat{a}_i \cdots a_k$, for $1 \leq i \leq k$. Recall that we are assuming that the input structure $\mathcal{A} = \langle \{0, 1, \cdots, n-1\}, R_1^{\mathcal{A}}, \cdots, R_p^{\mathcal{A}}, c_1^{\mathcal{A}}, \cdots, c_q^{\mathcal{A}} \rangle$ is coded as a bit string of length $I(n) = n^{r_1} + \cdots + n^{r_p} + q\lceil \log n \rceil$. It is clear that all of the consistent modifications above allow processor $a_1 \cdots a_k$ to test in constant time whether or not the relation $R(t_1, \cdots, t_r)$ holds, where R is an input or logical relation, and $t_j \in \{a_1, \cdots, a_k\} \cup \{c_j | 1 \leq j \leq q\}$.

$\square$

4 On the efficiency of the simulations

In this section we analyze the proof of Theorem 1.1 in more detail in order to give the following bounds for translating between CRAM and IND. After we prove Corollary 4.1, we discuss the cost of the simulation, and how these bounds can be improved. The proofs in this section involve counting how many variables are needed in various first-order formulas. This whole section should be omitted by the casual reader.

Corollary 4.1 *Let* CRAM$[t(n)]$-PROC$[p(n)]$ *be the complexity class* CRAM$[t(n)]$ *restricted to machines using at most* $O[p(n)]$ *processors. Let* IND$[t(n)]$-VAR$[v(n)]$ *be the complexity class* IND$[t(n)]$ *restricted to inductive definitions using at most* $v(n)$ *distinct variables. Assume for simplicity that the maximum size of a register word, and* $t(n)$ *are both* $o[\sqrt{n}]$, *and that* $\pi \geq 1$ *is a natural number. Then,*

$$\begin{aligned} &\text{CRAM}[t(n)]\text{-PROC}[n^{\pi}] \\ &\qquad\qquad \subseteq \quad \text{IND}[t(n)]\text{-VAR}[2\pi+2] \\ &\qquad\qquad\qquad\qquad \subseteq \quad \text{CRAM}[t(n)]\text{-PROC}[n^{2\pi+2}] \end{aligned}$$

Proof We prove these bounds using the following two lemmas.

Lemma 4.2 *If the maximum size of a register word, and* $t(n)$ *are both* $o[\sqrt{n}]$, *and if* M *is a* CRAM$[t(n)]$-PROC$[n^{\pi}]$ *machine, then the inductive definition of* VALUE *may be written using* $2\pi + 2$ *variables.*

Proof We write out the inductive definition of VALUE in enough detail to count the number of variables used:

$$\text{VALUE}(\overline{p}, t, x, r, b) \;\equiv\; Z \vee W \vee S \vee R \vee M \vee B \vee A\ ,$$

where the disjuncts have the following intuitive meanings:

Z: $t = 0$ and the initial value of r is correct.

W: $t \neq 0$ and the instruction just executed is WRITE, and the value of r is correct, i.e., unchanged unless r is Program_Counter.

S, R, M, B, A: Similarly for SHIFT, READ, MOVE, BLT, and, ADD or SUBTRACT, respectively.

It suffices to show that each disjunct can be written using the number of variables claimed. First we consider the disjunct Z. The only interesting part of Z is the case where r is "Processor". In this case we use the relation BIT to say that $b = 1$ if and only if the xth bit of $\overline{p}$ is 1. No extra variables are needed. Note that the number of free variables in the relation is $\pi + 1$ because we may combine the values t, x, r, and b into a single variable.

Next we consider the case of Addition. Recall that the main work is to express the carry bit:

$$C[A, B](x) \equiv (\exists y < x)[A(y) \wedge B(y) \wedge (\forall z. y < z < x) A(z) \vee B(z)] \ .$$

This definition uses two extra variables. Thus $\pi + 3 \leq 2\pi + 2$ variables certainly suffice. The cases S, M, and B are simpler.

The last, and most interesting case is R. Here we must say,

1. The instruction just executed is READ,

2. Register r is the Contents register,

3. There exists a processor $\overline{p'}$ and a time t' such that:

   (a) $t' < t$,
   (b) Address$(\overline{p'}, t')$ =Address$(\overline{p}, t)$,
   (c) VALUE$(\overline{p'}, t', x, r, b)$,
   (d) Processor $\overline{p'}$ wrote at time t',
   (e) For all $\overline{p''} < \overline{p'}$, if $\overline{p''}$ wrote at time t', then Address$(\overline{p''}, t') \neq$Address$(\overline{p'}, t')$,
   (f) For all t'' such that $t' < t'' < t$ and for all $\overline{p''}$, if $\overline{p''}$ wrote at time t'', then Address$(\overline{p''}, t'') \neq$Address$(\overline{p'}, t')$.

Let's count variables. On its face this formula uses three $\overline{p}$'s and three t's. However, two copies of each suffice. Observe that where we quantify $\overline{p''}$ in lines 3e and 3f, we no longer need $\overline{p}$, so we may use these variables instead. Similarly, when we quantify t'' on line 3f, we don't need $\overline{p''}$ so we may temporarily use one of its variables for t''. Finally, we would seem to need an extra variable to say "Address$(\overline{p''}, t'') \neq$Address$(\overline{p'}, t')$," in 3f. Here we use the fact that t is $o[\sqrt{n}]$, so t' and t'' can be coded into a single variable. Then with one more variable we can say that there exists a bit on which Address($\overline{p''}, t''$) and Address($\overline{p'}, t'$) disagree. Thus $2\pi + 2$ variables suffice as claimed. $\square$

The second lemma we need (Lemma 4.3) is a refinement of Lemma 3.2.

Lemma 4.3 *Let $\varphi(R, \overline{x})$ be an inductive definition of depth $d(n)$. Let k be the number of distinct variables including $\overline{x}$ occurring in φ. Then the relation defined by φ is also computable in* CRAM$[d(n)]$-PROC$[O[n^k]]$.

Proof This is very similar to the proof of Lemma 3.2. Let T be the parse tree of φ. The CRAM will have $n^k|T|$ processors: one for each value of the k variables and each node in T. Let δ be the depth of T. In rounds consisting of 3δ steps, the CRAM will evaluate an iteration of φ. Let $r =$ arity$(R) =$ the number of variables in $\overline{x}$; so $r \leq k$. The CRAM will have n^r bits of global memory to hold the truth value of $R_t = \varphi^t(\emptyset)$. It will use an additional $n^k|T|$ bits of memory to store the truth values corresponding to nodes of T. Thus $R_{d(n)}$, the least fixed point of φ, is computed in time $O[d(n)]$, using $O[n^k]$ processors, as claimed. $\square$

This completes the proof of Corollary 4.1. $\square$

The above proofs give us some information concerning the efficiency of our simulation of CRAMs with first-order inductive definitions. The main questions is, "Why is the number of variables needed to express a computation of n^π processors $2\pi + 2$, instead of π?" We discuss the multiplicative factor of two, and the additional two variables, respectively in the next two paragraphs.

We need the 2π term for two reasons: we must specify $\overline{p}$ and $\overline{p'}$ at the same time in order to say that their Address registers are equal; and we need to say that no lower numbered processor $\overline{p''}$ wrote into the same address as $\overline{p'}$. This term points out a difference between the CRAM model and the network described in Remark 3.3 that was used to simulate a FO$[t(n)]$ property. The factor of two would be eliminated if we adopted a weaker parallel machine model allowing only common writes[7], and such that the memory location accessed by a processor could be determined by a very simple computation on the processor number.

[7]See [6] for an earlier proof that a common write machine can simulate a CRAM with a linear increase in time and a squaring of the number of processors.

The additional two variables arise for various bookkeeping reasons. This term can be significantly reduced if we make the following two changes:

1. Rather than keeping track of all previous times, we can assume that every bit of global memory is written into at least every T time steps for some constant T.

2. The register size can be restricted to $O[\log n]$ so we need only $O[\log\log n]$ bits to name a bit of a word.

Remark 4.4 The above observations show that the relation between the number of variables needed to give an inductive definition of a relation, and the logarithm to the base n of the number of processors needed to quickly compute the relation are nearly identical. The cost of programming with first-order inductive definitions rather than CRAMs is theoretically very small. More work and even some experimentation must be done before one can say whether or not this will turn out to be a practical approach.

5 NC versus FO

In this section we relate the uniform NC circuit classes to $\mathrm{FO}[t(n)]$, and we derive a completely syntactic definition for circuit uniformity. We show that our definition is equivalent to the usual Turing machine-based definition in the range where the latter exists.

Let NC^i (respectively, AC^i) be the set of problems recognizable by a uniform sequence of polynomial size, bounded fan-in (respectively, unbounded fan-in) boolean circuits of depth $\log^i n$. Let $\mathrm{NC} = \mathrm{AC} = \bigcup_i \mathrm{NC}^i$. Ruzzo characterized these uniform circuit classes in terms of alternating Turing machines:

Fact 5.1 ([15]) *For* $i \geq 1$,

$$\begin{aligned}\mathrm{NC}^i &= \mathit{ASPACE\text{-}TIME}[\log n, \log^i n]\ ,\\ \mathrm{AC}^i &= \mathit{ASPACE\text{-}ALT}[\log n, \log^i n]\ .\end{aligned}$$

Ruzzo and Tompa proved the following relationship between the uniform AC classes and the CRAM :

Fact 5.2 ([17]) *For* $i \geq 1, \mathrm{AC}^i = \mathrm{CRAM}[\log^i n]$.

The following corollary of Theorem 1.1 and Fact 5.2 shows that the uniformity condition for the AC^i circuit classes can be described in a syntactic way. A first-order sentence iterated $t(n)$ times is also an AC circuit "iterated" $t(n)$ times. Thus we no longer need to mention machines when discussing uniform circuit complexity.

Corollary 5.3 *For $i \geq 1$, $\mathrm{AC}^i = \mathrm{FO}[\log^i n]$.*

Before now there was no satisfactory definition for uniform AC^0. It is easy to see that a first-order sentence corresponds to a particularly simple sequence of AC^0 circuits. Each quantifier $\exists x$ (respectively, $\forall x$) is just an n-ary "or" (respectively, "and"). In [11] we showed that an appropriate way to make first-order sentences nonuniform is to add an arbitrary new logical relation. The following fact says that nonuniform AC^0 is equal to nonuniform FO.[8]

Fact 5.4 ([11]) *Given a problem S and an integer $d > 1$ the following are equivalent:*

1. *S is recognized by a sequence of depth $d+1$, polynomial-size circuits, with bounded fan-in at the bottom level.*

2. *There exists a new logical relation $R \subset \mathbf{N}^a$ and a first-order formula φ in which R occurs such that φ expresses S. The formula φ contains d alternating blocks of quantifiers.*

In view of the above results, we propose the following:

Definition 5.5 Let (uniform) $\mathrm{AC}^0 \stackrel{\mathrm{def}}{=} \mathrm{FO}[1] = \mathrm{CRAM}[1]$.

Since we first made this suggestion, much evidence concerning the appropriateness of Definition 5.5 has appeared. In particular, see [1] for a study of low-level uniformity. It is shown there that FO is equal to deterministic log time uniform AC^0.

In [11] we introduced the notion of first-order translations. These reductions consist of a fixed first-order formula translating all instances of one problem to instances of another. (First-order translations are interpretations between theories, cf. [4], that are also reductions in the complexity

[8]In [17] Stockmeyer and Vishkin showed that nonuniform AC^0 is equal to constant time on a nonuniform CRAM. This, together with Fact 5.4, gives a nonuniform version of Theorem 1.1.

theoretic sense.) It follows from Definition 5.5 that first-order translations are exactly uniform AC^0 reductions.

One way to evaluate the appropriateness of Definition 5.5 is to examine examples of AC^0 reductions in the literature and see whether or not they can be made uniform. Of those we have considered, the answer is yes, with the following interesting exception. (The UGAP problem is the set of undirected graphs for which there exists a path from vertex 0 to vertex $n-1$.)

Fact 5.6 *[3].* UGAP *is nonuniform* AC^0 *reducible to* UNDIR-CYCLE.

Now UNDIR-CYCLE is in DSPACE$[\log n]$ [8], but UGAP is not known to be in DSPACE$[\log n]$. Of course,

Remark 5.7 If UGAP is uniform AC^0 reducible to UNDIR-CYCLE, then UGAP is in DSPACE$[\log n]$.

We mention one more interesting justification of Definition 5.5. In [3] it is shown that the obvious bounds,

$$\text{nonuniform NC}^i \;\subseteq\; \text{nonuniform AC}^i$$

can be improved to

Fact 5.8 ([3])

$$\text{nonuniform NC}^i \;\subseteq\; \text{nonuniform AC-DEPTH}[\log^i n/\log\log n]\,.$$

When $i=1$ this bound is optimal because nonuniform AC-DEPTH$[\log n/\log\log n]$ is necessary for Parity [18]. We next show that the same bound holds in the uniform case:

Theorem 5.9 *For $t(n) \geq \log n$,*

$$\text{ASPACE}[\log n]-\text{TIME}[t(n)] \subseteq \text{FO}[t(n)/\log\log n]\ .$$

Proof This is a $\log\log n$ factor improvement of Theorem B.3 in [9]. There we showed how to code a log space Turing machine configuration using a constant number of variables, as well as how to write the predicate $M_1(\bar{x},\bar{y})$, meaning that $\langle\bar{x},\bar{y}\rangle$ is a valid move of the given alternating Turing machine. We could then inductively define the predicate $\text{Accept}_t(\bar{x})$, meaning that the configuration $\bar{x}$ leads to acceptance in the sense of alternating Turing machines in t steps:

$$\text{Accept}_t(\bar{x}) \equiv (\exists \bar{y}.M_1(\bar{x},\bar{y}))(\forall \bar{z}.M_2)\text{Accept}_{t-1}(\bar{z}) \ ,$$

where $M_2 \equiv (\bar{z} = \bar{y}) \vee$ ("$\bar{x}$ is universal" $\wedge\, M_1(\bar{x},\bar{z})$).

To improve this simulation by a $\log\log n$ factor, observe that a list of which existential moves to make in the event of each possible sequence of $(\log\log n)/2$ universal moves can be given in $\log n$ bits. Thus we can write,

$$\text{Accept}_t(\bar{x}) \equiv (\exists e \forall u)(\exists \bar{z})(R \wedge \text{Accept}_{t-\log\log n}(\bar{z})) \ , \tag{1}$$

where R says that $\bar{z}$ follows from $\bar{x}$ in the $\log\log n$ moves determined by e and u.

Now it is easy to write an inductive definition of R whose depth is $\log\log n$. This definition uses the BIT predicate to decode from e and u which of the possible two moves the Turing machine makes at each of the $\log\log n$ steps. The simultaneous inductive definition of Accept is given in Equation 1. Obviously its depth is $\log n/\log\log n$. $\square$

Corollary 5.10 *For* $i \geq 1$, $\text{NC}^i \subseteq \text{FO}[\log^i n/\log\log n]$.

6 The polynomial-time hierarchy

In second-order logic we have first-order logic, plus new relation variables over which we may quantify. Let A_i^j be a j-ary relation variable. Then $(\forall A_i^j)\varphi$ means that for all choices of j-ary relation A_i^j, φ holds. It is well known that second-order formulas may be transformed into prenex form, with all second-order quantifiers in front. Let SO be the set of second-order expressible properties, and let (SO $\exists$) be the set of second-order properties that may be written in prenex form with no universal second-order quantifiers. Fagin gave the following interesting characterization of nondeterministic polynomial-time (NP) in terms of logical expressibility:

Fact 6.1 ([5]) (SO$\exists$) = NP .

A few years later, when he defined the polynomial-time hierarchy (PH), Stockmeyer showed that it coincided with the set of second-order expressible properties:

Fact 6.2 ([16]) PH = SO.

As a corollary to Fact 6.2 and Theorem 1.1, we obtain the following characterization of PH as a parallel complexity class:

Corollary 6.3 PH *is equal to the set of properties checkable by a* CRAM *using exponentially many processors and constant time*[9]*:*

$$\mathrm{PH} \;=\; \bigcup_{k=1}^{\infty} \mathrm{CRAM}[1]\text{-}\mathrm{PROC}[2^{n^k}]\ .$$

Proof The inclusion SO $\subseteq$ CRAM[1]-PROC[$2^{n^{O[1]}}$] follows just as in the proof of Lemma 3.2. A processor number is now large enough to give values to all the relational variables as well as to all the first-order variables. Thus, as in Lemma 3.2, the CRAM can evaluate each first or second-order quantifier in three steps.

The inclusion CRAM[1]-PROC[$2^{n^{O[1]}}$] $\subseteq$ SO follows just as in the proof of Lemma 3.1. The only difference is that we use second-order variables to specify the processor number. □

7 Conclusions

To recapitulate, we have shown that parallel time has a simple mathematical definition: the minimal parallel time needed to compute a property using at most polynomially many processors is equal to the minimum depth of a first-order inductive definition of the property. Furthermore, the number of processors needed by the CRAM is closely tied to the number of variables needed in the inductive definition. We have also given purely syntactic definitions for uniformity of the circuit complexity classes AC^i, $i \geq 0$. Finally, we have given a striking, new characterization of the polynomial-time hierarchy. We believe that these results help to explain the nature of parallel complexity and will lead to an improved understanding of the subject.

There is much work to be done. The following general directions suggest themselves:

[9] Up to this point we had been assuming for notational simplicity that a CRAM has at most polynomially many processors. However, the class CRAM[$t(n)$]-PROC[$p(n)$] still makes sense for numbers of processors $p(n)$ that are not polynomially bounded.

1. This paper provides a new way to think about parallel programming. The programmer provides efficient inductive definitions of the problem to be solved. Our simulation results then automatically give an efficient implementation on a CRAM. Much work is needed to explore whether or not this approach is practical.

2. We have given characterizations of parallel time and number of processors in terms of the depth and number of variables in inductive definitions. One should now develop upper and lower bounds on these parameters for all sorts of problems. We also feel that the analysis of the simulation in §4 can and should be improved.

3. There are many fascinating questions concerning uniformity and the power of precomputation. We hope that the notion of syntactic uniformity of circuits will help researchers determine when precomputation/nonuniformity can help; or, to prove lower bounds on what can be done by uniform circuits and formulas.

Acknowledgments. Thanks to Steve Cook, Steven Lindell, Ruben Michel, and Larry Ruzzo, who contributed comments and corrections to previous drafts of this paper.

References

[1] D. M. BARRINGTON, N. IMMERMAN, AND H. STRAUBING, *On uniformity within* NC^1, Proc. 3rd Annual Symposium on Structure in Complexity Theory (1988), pp. 47-59.

[2] P. BEAME, *Limits on the power of concurrent-write parallel machines,* Proc. 18th ACM Symposium on Theory of Computing (1986), pp. 169-176.

[3] A. CHANDRA, L. STOCKMEYER, AND U. VISHKIN, *Constant depth reducibility,* SIAM J. of Comput. 13 (1984), pp. 423-439.

[4] H. ENDERTON, *A Mathematical Introduction to Logic,* Academic Press, New York, 1972.

[5] R. FAGIN, *Generalized first-order spectra and polynomial-time recognizable sets,* in Complexity of Computation, R. Karp, ed., SIAM-AMS Proc., 7 (1974), pp. 27-41.

[6] F. FICH, P. RAGDE, AND A. WIGDERSON, *Relations between concurrent-write models of parallel computation,* Proc. 3rd Annual

ACM Symposium on Principles of Distributed Computing (1984), pp. 179-189.

[7] D. HAREL AND D. KOZEN, *A programming language for the inductive sets, and applications,* Proc. 9th International Colloquium on Automata Languages, and Programming, Springer-Verlag Lecture Notes in Computer Science, 140 (1982), pp. 313-329.

[8] J.-W. HONG, *On some deterministic space complexity problems,* SIAM J. Comput., 11 (1982), pp. 591-601.

[9] N. IMMERMAN, *Upper and lower bounds for first-order expressibility,* J. Comput. System. Sci., 25 (1982), pp. 76-98.

[10] ———, *Relational queries computable in polynomial time,* Inform. and Control, 68 (1986), pp. 86-104. A preliminary version of this paper appeared in Proc. 14th ACM Symposium on Theory of Computing (1982), pp. 147-152.

[11] ———, *Languages that capture complexity classes,* SIAM J. Comput., 16 (1987), pp. 760-778. A preliminary version of this paper appeared in Proc. 15th ACM Symposium on Theory of Computing (1983),pp. 347-354.

[12] ———, *Expressibility as a complexity measure: Results and directions,* Proc. 2nd Annual Symposium on Structure in Complexity Theory (1987), pp. 194-202.

[13] ———, *Descriptive and computational complexity,* Proc. 1988 AMS Short Course in Computational Complexity Theory, to appear.

[14] Y.N.MOSCHOVAKIS, *Elementary Induction on Abstract Structures,* North Holland, Amsterdam, 1974.

[15] L. RUZZO, *On uniform circuit complexity,* J. Comput. System. Sci., 21 (1981), pp. 365-383.

[16] L. STOCKMEYER, *The polynomial-time hierarchy,* Theoretical Comput. Sci., 3 (1977), pp. 1-22.

[17] L. STOCKMEYER AND U. VISHKIN, *Simulation of parallel random access machines by circuits,* SIAM J. of Comput. 13 (1984), pp. 409-422.

[18] A. C.-C. YAO, *Separating the polynomial-time hierarchy by oracles,* Proc. 26th Annual IEEE Symposium on Foundations of Comput. Sci. (1985), pp. 1-10.