Parallel RAMs with Owned Global Memory and Deterministic Context-Free Language Recognition

Patrick W. Dymond
Walter L. Ruzzo

Technical Report UW-CSE-97-02-03
February, 1997

Also York University Computer Science Technical Report CS-97-02.

Department of Computer Science & Engineering
University of Washington
Seattle, WA 98195

Parallel RAMs with Owned Global Memory and Deterministic Context-Free Language Recognition

Patrick W. Dymond*‡ Walter L. Ruzzo†‡

February, 1997

Abstract

We identify and study a natural and frequently occurring subclass of Concurrent-Read, Exclusive-Write Parallel Random Access Machines (CREW-PRAMs). Called Concurrent-Read, *Owner*-Write, or CROW-PRAMs, these are machines in which each global memory location is assigned a unique "owner" processor, which is the only processor allowed to write into it. Considering the difficulties that would be involved in physically realizing a full CREW-PRAM model, it is interesting to observe that in fact, most known CREW-PRAM algorithms satisfy the CROW restriction or can be easily modified to do so. This paper makes three main contributions. First, we formally define the CROW-PRAM model and demonstrate its stability

*Department of Computer Science, CCB126, York University, Toronto, Canada M3J 1P3; `dymond@cs.yorku.ca`.

†Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350; `ruzzo@cs.washington.edu`.

‡Parts of this work were done at the Department of Computer Science and Engineering, University of California, San Diego, and at the Computer Science Department, University of Toronto, whose hospitalities are gratefully acknowledged. Supported in part by the Natural Sciences and Engineering Research Council and by NSF grants ECS-8306622, DCR-8604031, CCR-8703196, CCR-9002891, and NSF/DARPA grant CCR-8907960.

under several definitional changes. Second, we precisely characterize the power of the CROW-PRAM by showing that the class of languages recognizable by it in time $O(\log n)$ is exactly the class LOGDCFL of languages log space reducible to deterministic context free languages. Third, using the same basic machinery, we show that the recognition problem for deterministic context-free languages can be solved in time $O(n^{1+\epsilon}/S(n))$ for any $\epsilon > 0$ and any $\log^2 n \leq S(n) \leq n$ on a deterministic auxiliary pushdown automaton having a $\log n$ space work tape, pushdown store of maximum height $S(n)$, and random access to its input tape. These results extend and unify work of von Braunmühl, Cook, Mehlhorn, and Verbeek; Klein and Reif; and Rytter.

1 Introduction and Related Work

There is now a fairly large body of literature on parallel random access machine (PRAM) models and algorithms. There are nearly as many definitions of this model as there are papers on the subject. All agree on the general features of such models — there is a collection of more or less ordinary sequential processors with private, local memories, that all have access to a shared global memory. The model is synchronous — in each time unit, each processor executes one instruction. There is much more diversity regarding other features of the model. For example, there are differences as to whether the model has single- or multiple-instruction streams, how many processors there are, how they are numbered, how they are activated, what instruction set they have, what input convention is used, and how simultaneous read or write requests to a single global storage location are arbitrated. Most of these variations make little or no difference in the power of the model.

Two features seem to have a substantial impact on the power of the model. One is uniformity. In general, we only consider uniform models in this paper, i.e., ones where a single program suffices for all input lengths, and where a single processor is initially active, creating other processors as desired. The second sensitive feature is arbitration of memory access conflicts. Two main variants have been most intensively studied. Following the nomenclature introduced by Vishkin [37], the CRCW (Concurrent-Read, Concurrent-Write) PRAM allows memory access conflicts. All processors reading a given location in a given step receive its value. Among all processors writing to a given location in a given step, one is allowed to succeed, e.g., the one with the lowest processor number. (Other resolution rules for write conflicts have been proposed. All are known to be equivalent in power up to constant factors in running time, and polynomial factors in number of processors and global memory size, although the models are separated if processors and memory are more tightly constrained.)

In the CREW (Concurrent-Read, Exclusive-Write) model, concurrent reads are allowed, as above, but concurrent writes are not. CREW algorithms must arrange that no two processors attempt to write into the same global memory location at the same time.

In this paper we introduce a third variant, argue that it is a more "natural" model than the CREW PRAM, and give a surprising characterization of its power. There are several reasons to study this restriction of the CREW-PRAM. The CREW-PRAM model has been criticized for being too powerful to serve as a realistic model of physically realizable parallel machines due to its "unbounded fanin." Anderson and Snyder [1] point out that the two-stage programming process of first using the CREW-PRAM model to develop a straightforward fully parallel algorithm (e.g., for the "or" of n bits), and then emulating this algorithm on a physically realizable network, could lead to a sub-optimal algorithm ($\Theta((\log n)^2)$ for the above example). Nevertheless the CREW-PRAM has arguably been the most popular theoretical model for the design, specification and analysis of parallel algorithms, due principally to the simplicity and usefulness of the global memory model for programmers. It is useful therefore to consider the power of the more restricted CROW-PRAM model, in order to understand its feasibility as a model for parallel programming. As noted above, most CREW-PRAM algorithms are in fact CROW-PRAM algorithms, or can be easily modified to be so.

How can a CREW-PRAM algorithm ensure that it is obeying the Exclusive-Write restriction? With two exceptions discussed below, all CREW-PRAM algorithms we have considered achieve, or can be easily modified to achieve, write exclusion by the following simple stratagem: each global memory location is "owned" by one processor, which is the only processor ever allowed to write into that cell. Further, the mapping between global memory addresses and processor numbers is easy to compute, so that each processor has no difficulty in determining which cells it owns. For example, processor p might own the block of k consecutive cells beginning at global memory address kp. We call this the *Owner-Write* restriction, and call PRAMs that obey this restriction *Concurrent-Read, Owner-Write PRAMs*, or *CROW-PRAMs*. The ownership restriction seems to be a very natural framework in which to design exclusive-write algorithms. Similar but not identical notions of "ownership" have appeared in the earlier lower bound work of Cook, *et al.* [7], and have also proven useful in practice for certain cache coherence protocols. (See, e.g., Archibald and Baer [2].) In many current architectures of parallel systems, the machines provide a global memory programming model, implemented using physical hardware in which every memory cell is local to some processor. Caching or other techniques are used to ameliorate the cost of access to non-local memory. If non-local writes are prohibited, the necessary cache coherence algorithms are simplified. In fact, a positive solution to the CROW versus CREW problem discussed in Section 3 would presumably suggest an interesting new approach to the cache coherence problem.

We give a precise definition of the CROW-PRAM model in Section 2 below. The main goal of this paper is to investigate the power of the CROW-PRAM. Unexpectedly, this question turns out to be intimately related to the complexity of deterministic context-free language (DCFL) recognition.

The recognition problem for a deterministic context-free language L is to decide, given a word x, whether $x \in L$. The sequential complexity of this problem has been well-studied, and there are many practical sequential algorithms for solving it in space and time $O(n)$. The small-space and parallel time complexities of the problem are less well-understood. Two main results in these areas are by von Braunmühl, Cook, Mehlhorn, and Verbeek [5, 38], and by Klein and Reif [20].

Cook [5] presents a sequential algorithm for the DCFL recognition problem that runs in polynomial time on a Turing machine using only polynomial in $\log n$ space. This result has been improved by von Braunmühl *et al.* [38], who give Turing machine algorithms with optimal time-space product for any space bound in the range from $(\log n)^2$ to n.

Building somewhat on the ideas of [5, 38], Klein and Reif [20] present an $O(\log n)$ time CREW-PRAM algorithm for DCFL recognition. (It is known that results of Stockmeyer and Vishkin [35] can be combined with the algorithm of Ruzzo [32] to yield an $O(\log n)$ time algorithm for general CFL recognition, but only on the more powerful CRCW-PRAM model.)

Our main result is the following characterization of CROW-PRAMs.

Theorem 1 *A language L is accepted by a CROW-PRAM in $O(\log n)$ time if and only if L is log-space reducible to a DCFL.*

The class LOGDCFL of languages log-space reducible to DCFLs was first defined and studied by Sudborough [36], who showed that it is equal to the class of languages recognizable in polynomial time by log-space bounded deterministic auxiliary pushdown automata (DauxPDAs), defined by Cook [4].

Our result appears to be the first to precisely characterize a parallel time complexity class (up to constant factors) in terms of a sequential one. For example, Sudborough's "hardest DCFL" [36] provides a natural example of a problem complete for CROW-PRAM time $O(\log n)$. Complete problems have been discovered by Chandra and Tompa for CRCW-PRAM time classes [3]. We know of no analogous natural problems that are complete for CREW-PRAM time classes. Following an earlier version of our paper [12], Lange and Niedermeier [22] established characterizations of other PRAM variants in terms of sequential complexity classes.

We use the DCFL characterization to demonstrate the stability of CROW-PRAM complexity classes under definitional changes. For example, it follows from the DCFL simulation that a CROW-PRAM can be simulated without time loss by a parallel machine on which there is no global memory, but each processor contains a single externally visible register, that may be read (but not written) by any other processor. This model seems to be closer to the way that some parallel machines have actually been constructed than models with an independent global memory not associated with any processor.

The DCFL recognition algorithms of von Braunmühl *et al.* [38] and Klein and Reif [20] are difficult ones, and use superficially different approaches. The third goal of this paper is to provide a unified approach to both problems, which, although based on both, we believe to be simpler than either.

We obtain both a small time parallel algorithm and a small space sequential algorithm for DCFL recognition using the same basic approach. The small space algorithm provides an improvement to a result by Rytter [33], and a technical refinement to the optimal results of von Braunmühl *et al.* [38]. Rytter had shown, using a sequential implementation of [20], that it is possible to obtain a polynomial time, $O(\log^2 n)$ space algorithm for DCFL recognition using space mainly as a pushdown store (more precisely, a $\log n$ space DauxPDA with an $O(\log^2 n)$ bounded pushdown), rather than unrestricted $O(\log^2 n)$ space as in [38]. We improve these results by performing our simulation on a DauxPDA (like Rytter) while attaining a time-space product similar to that of von Braunmühl *et al.*

Section 2 presents the CROW-PRAM model, and discusses variations in the definition. Section 3 presents its simulation by deterministic auxiliary pushdown automata, establishing CROW-PRAM-TIME($\log n$) $\subseteq$ LOGDCFL. Section 4 introduces some definitions and notation needed in our DCFL recognition algorithm. Section 5 presents a high level description and correctness proof of the DCFL recognition algorithm. Section 6 discusses CROW-PRAM implementation of the algorithm, establishing the other inclusion needed for Theorem 1, i.e., LOGDCFL $\subseteq$ CROW-PRAM-TIME($\log n$). Finally, Section 7 refines the simulation of Section 5 to obtain a faster sequential algorithm than that obtained by combining the CROW-PRAM algorithm of Section 6 with the general simulation of Section 3.

Further work involving the owned global memory concept in PRAMs has appeared following a preliminary version of this paper [12]. Fich and Wigderson give a lower bound separating EROW and CROW PRAMs [14]. Rossmanith introduces and studies Owner Read, Owner Write PRAMs, showing, for example, that they can do list ranking in $O(\log n)$ time [31]. Nishimura considers the owner concept in CRCW-PRAMs [29]. Niedermeier and Rossmanith [27, 26] have considered the owner concept with other PRAM variants. Lin, *et al.* show that CROW-PRAMs are sufficiently powerful to execute a variant of Cole's parallel merge sort algorithm in time $O(\log n)$ [23]. Work on further restrictions of the CROW-PRAM model by Lam and Ruzzo [21] and Dymond, *et al.* [11] is described at the end of section two.

2 Definition of CROW-PRAMs

We start by defining the CREW-PRAM model we will use. As mentioned above, most of the details of the definition are not critical. For specificity we use the definition of Fortune and Wyllie [15] (called simply a P-RAM there) which has: an unbounded global memory and an unbounded set of processors, each with an accumulator, an instruction counter and an unbounded local memory. Each memory cell can hold an arbitrary non-negative integer. The instruction repertoire includes indirect addressing, load, store, add, subtract, jump, jump-if-zero, read, fork, and halt. The input is placed in a sequence of special read-only registers, one bit per register. The read instruction allows any processor to read any input bit; concurrent reads are allowed. A fork instruction causes a new processor to be created, with all local memory cells zero, and with its accumulator initialized to the value in the accumulator of its creator. Initially, one processor is active, with its local memory zero, and the length of the input given in its accumulator. The model accepts if the initially active processor halts with a one in its accumulator. It rejects if two processors attempt to write into the same global memory location at the same time.

These CREW-PRAMs do not have "processor numbers" or "IDs" as a built-in concept, but we will need them. We adopt the following processor numbering scheme. The (unique) processor active initially is numbered 0; the first child processor created by processor i will be numbered $2i+1$, its second $2(2i+1)$, $\ldots$, and its $k^{th}, k \geq 1$ will be numbered $2^{k-1}(2i+1)$. This corresponds to the natural embedding of an arbitrary tree (the processor activation tree) into a binary tree by the rule "eldest child becomes right child, next younger sibling becomes left child." Reverse-preorder traversal of the activation tree and the binary tree are identical. As we will see, many other numbering schemes will also work; this one is fairly natural. Processors do not automatically "know" their number, but it is easy to program them to compute it, if needed.

Definition: A *CROW-PRAM algorithm* is a CREW-PRAM algorithm for which there exists a function $owner(i, n)$, computable in deterministic space $O(\log n)$, such that on any input of length n processor p attempts to write into global memory location i only if $p = owner(i, n)$.

The intuitive definition given earlier said that the owner function should be "simple". We have particularized this by requiring that it be log-space computable and that it be oblivious, i.e., independent of the input, except for its length. We have not required that the model detect ill-behaved programs, i.e., ones that attempt global writes in violation of the ownership constraint. Such programs simply are not CROW programs. These seem to be natural choices, but we will also show that our main results are fairly insensitive to these issues. We could generalize the model in any or all of the following ways:

G1. Allow the owner function to depend on the input.

G2. Allow the owner function to depend on time.

G3. Allow "bounded multiple ownership", i.e., $owner(i, n)$ is a set of size $O(1)$ of processor numbers.

G4. Allow ill-behaved programs, by defining the model to halt and reject if an attempted write violates the ownership constraint.

G5. Allow any processor numbering scheme that gives processors unique numbers and allows one to compute in logarithmic space the parent of a given processor p, the number of older siblings it has, and the number of its k^{th} child.

G6. Allow the owner, parent, and sibling functions above to be computable by a deterministic log-space auxiliary pushdown automaton that runs in polynomial time.

Alternatively, we could restrict the model in any or all of the following ways:

R1. Require that the owner function be the identity — $owner(i, n) = i$. This is equivalent to saying that the machine has no global memory; instead it is a collection of processors each with a private local memory and one globally readable "communications register"

R2. Require that processors use only O(1) local memory locations.

R3. Require that the machine be *write-oblivious*, i.e., the times and locations of writes to global memory are independent of the input, except for its length.

One consequence of our results is that CROW-PRAMs, even ones satisfying only the relatively weak conditions G1-G6, can be simulated by CROW-PRAMs satisfying the strict conditions R1-R3, with only a constant factor increase in time and a polynomial increase in number of processors.

Is it possible that CREW- and CROW-PRAMs have equivalent power? On the positive side, conditions G1-G6 are fairly generous. It is difficult to imagine a protocol by which a PRAM algorithm could achieve write-exclusion that would not be covered by these. For example, note that a general CREW-PRAM algorithm can be considered to be a CROW-PRAM algorithm where the owner function is allowed to be input- and time-dependent (conditions G1 and G2 above) and in some sense computable by a CREW-PRAM in real-time. We know that, say, CREW-PRAM time $O(\log n)$ can be simulated by a logarithmic space deterministic auxiliary pushdown automaton that runs in time $n^{O(\log n)}$, so real-time CREW-PRAM computable functions may not be that different from $n^{O(1)}$ DauxPDA computable ones. Thus it seems possible that time on CROW-PRAMs and CREW-PRAMs might be identical. At least, this provides some intuitive support for the empirical observation that most known CREW-PRAM algorithms are CROW-PRAM algorithms.

In one context, we know the two models *are* equivalent. Following the appearance of an extended abstract of this paper [12], Ragde (personal communication; see also Fich [13], Nisan [28]) observed that *nonuniform* CROW-PRAMs, i.e., ones having arbitrary instructions, exponentially many processors initially active, and allowing different programs for each value of n, running in time t are equivalent to Boolean decision trees of depth 2^t. Nisan [28] established that for any set recognized by a (nonuniform) CREW-PRAM in time $t(n) = O(\log n)$, for each n there is a equivalent Boolean decision tree problem of depth $2^{t(n)}$. Taken together these results show time on the two models is the same up to a constant factor in the nonuniform setting. This leaves open the stronger conjecture that any set recognized by a CREW-PRAM in time $\log n$ can be recognized on a CROW-PRAM in time $O(\log n)$, both of the ordinary, uniform variety and both using polynomially many processors. Note that Nisan's simulation of CREW by CROW uses nonuniformity in a fundamental way and uses $2^{2^{t(n)}}$ initially active processors, and that in his nonuniform model *all* languages are recognizable in $O(\log n)$ steps.

In one restricted setting we know the two (uniform or nonuniform) models to be different. Suppose processors 1 through n are active, each knows one bit b_i, and we want to compute the "or" of these bits, *given that at most one b_i is 1.* A CREW-PRAM can solve this in one step: any processor having a 1 bit writes it into global location 0. No write-conflict can happen since there is at most one 1 bit. However, Marc Snir (personal communication) has shown that a CROW-PRAM requires $\Omega(\log n)$ steps to solve this problem from the same initial state.

Snir's result does not settle the general question, however. The problem discussed above is defined only when at most one input bit is one. (This has been called a "partial domain" by Fich, in contrast to the more usual situation where an algorithm is required to produce a correct answer on *all* n-bit input sequences.) We know from the results of Cook, *et al.* [7] that even a CREW-PRAM requires time $\Omega(\log n)$ to test whether its input contains at most one 1 bit. Conceivably, a CREW algorithm that exploited something like Snir's "or" could always be transformed into a CROW algorithm by using this "preprocessing" time to better advantage.

The *only* full domain problem known to us where (uniform) CREW-PRAMs seem more powerful that CROW-PRAMs is the recognition problem for *unambiguous* context-free languages. For this problem Rytter [34] has given an $O(\log n)$ CREW-PRAM algorithm that appears to use the power of non-owner exclusive writes in a fundamental way. Loosely speaking, it seems that the unambiguity of the underlying grammar allows one to repeatedly exploit a feature like Snir's "or".

While CROW-PRAMs appear to be nearly as powerful as CREW-PRAMs, it is interesting to compare them to a possibly weaker parallel model, the *parallel pointer machine* of Dymond and Cook [10]. PPMs consist of an unbounded pool of finite-state transducers, each with a finite set of pointers to other processors. A PPM operates by sensing the outputs of its neighboring processors, and moving its pointers to other processors adjacent to its current neighbors. Cook proposed such a model as an example of the simplest possible parallel machine with "variable structure" [6].

Lam and Ruzzo [21] establish that time on PPMs is linearly related to time on a restricted version of the CROW-PRAM, on which doubling and adding one are the only arithmetic operations permitted. (In fact, they also showed a simultaneous linear relationship between the amounts of hardware used on the two machines.) Our conjecture that the CROW-PRAM's ability to access two-dimensional arrays in constant time cannot be directly emulated on a CROW-PRAM whose arithmetic capability is so limited has been proved recently by Dymond, *et al.* [11]. Since two-dimensional arrays appear to play an important part in the DCFL simulation algorithm of Section 6, this suggests that quite different techniques would be needed to recognize DCFLs in time $O(\log n)$ on the PPM, if this is indeed possible. An analogous nonconstant lower bound on two dimensional array access was proved for sequential unit cost successor RAMs by Dymond [9].

3 Simulation of CROW-PRAMs by DauxPDAs

In this section we will prove the first half of Theorem 1, namely:

Theorem 2 *Any set recognized in time $O(\log n)$ on a CROW-PRAM is in LOGDCFL.*

Recall that LOGDCFL is the class of languages log space reducible to deterministic context-free languages. Sudborough [36] defined the class, and characterized it as the set of languages recognized in polynomial time on a logarithmic space deterministic auxiliary pushdown automaton.

The main construction is similar to analogous ones given by Pratt and Stockmeyer [30], Fortune and Wyllie [15], and Goldschlager [17] showing that PRAM time $\log n$ is contained in $DSPACE(\log^2 n)$. We define three mutually recursive procedures:

state(t, p) returns the state of processor p at time t, i.e., after the t^{th} instruction has been executed.

local(t, p, i) returns the contents of location i of the local memory of processor p at time t.

global(t, i) returns the contents of global memory location i at time t.

Each depends only on the values of these procedures at time $t-1$, so the recursion depth will be at most t. Furthermore, each procedure will require only $O(\log n)$ bits of local storage, so by well-known techniques these procedures can be implemented on a logarithmic space deterministic auxiliary PDA whose pushdown height is at most $O(\log^2 n)$. This much of the proof is essentially the same as in [15, 17, 30]. The main novelty with our proof is that our algorithm runs in polynomial time, rather than time $n^{\log n}$ as in the earlier results. This is possible because the owner function allows us in *global*(t, i) to directly identify the only possible writer of global memory location i at time $t-1$. This allows each of our procedures to make only $O(1)$ recursive calls per invocation, which gives a polynomial running time. If we were simulating a general CREW-PRAM algorithm, it would appear necessary to check *all* processors at time $t-1$ to see whether any of them wrote into i, and if so, whether more than one of them did. This appears to require more than polynomial time.

Extensions to these basic procedures to accommodate generalizations G1-G6 are quite direct, except for G4, ill-behaved programs. G4 is also possible, but more delicate, since in effect we must check at each step that *none* of the many non-owners attempts to write to a global cell, while maintaining the property that our algorithm makes only $O(1)$ recursive calls per invocation. (It is possible that a similar generalization of the CREW model would increase its power.)

Proof of Theorem 2: Detailed descriptions of the three procedures follow. A typical PRAM instruction is “global indirect store l”, whose meaning is “store the accumulator into the global memory location whose address is given by the contents of local memory location l”. We will not describe the rest of the PRAM's instruction set in great detail; see Fortune and Wyllie [15].

The *state* of processor p at time t is an ordered pair containing the *instruction counter*, and the contents of the *accumulator* of p at the end of the t^{th} step. We define three auxiliary functions *accumulator*(S), *instruction-counter*(S), and *instruction*(S), that, for any state S, give the accumulator portion of S, the instruction counter portion of S, and the instruction pointed to by the instruction counter of S, respectively. Assume that a value of 0 in the instruction counter designates a “halt” instruction, which by convention will be the instruction "executed" in each step before processor p is activated and after it has halted. Also, assume that $instruction(S)$ will be a "halt" instruction if it is not otherwise defined, e.g., after a jump to a location beyond the end of the program. It is convenient to assume that the local memory of a processor is set to zero as soon as it halts, but its accumulator retains its last value. We assume that processor 0 initially executes instruction 1, and that a processor activated by a "fork l" instruction initially executes instruction l. We also assume that each processor maintains in local memory location 0 a count of the number of "fork" instructions it has executed. (This count should be initially 0, and is incremented immediately after each "fork" is executed.) It is easy to modify any PRAM algorithm to achieve this. We also use two functions $parent(p)$ and $sibling\text{-}count(p)$ that, for any processor number p, return the processor number of the parent of p, and the number of older siblings of p, respectively. For the processor numbering scheme we have chosen these functions are very easy to compute. Namely, if k is the largest integer such that p is evenly divisible by 2^k, then $sibling\text{-}count(p) = k$, and $parent(p) = \lfloor p/2^{k+1} \rfloor$.

procedure *Simulate-CROW-PRAM*
comment: Main Program.
begin
  let $T = c \lceil \log n \rceil$ **comment:** An upper bound on the running time of the PRAM.
  if $state(T, 0) = (0, 1)$ **then accept**
end

function $global(t, i)$
comment: Returns the contents of global memory location i at time t.
begin
  if $t = 0$ **then return** 0
  $S := state(owner(i), t - 1)$
  if $instruction(S) =$ "global indirect store l" **and**
   $local(t - 1, owner(i), l) = i$
    then return $accumulator(S)$
    else return $global(t - 1, i)$
end

```
function local(t, p, i)
comment: Return the contents of local memory location i of proces-
  sor p at time t.
begin
    if t = 0 then return 0
    S := state(t − 1, p)
    case instruction(S):
        "halt"                    : return 0
        "local store i"           : return accumulator(S)
        "indirect local store l"  : if i = local(t − 1, p, l)
                                        then return accumulator(S)
    end
    return local(t − 1, p, i)
  end
```

```
function state(t, p)
comment: Return the state of processor p at time t.
begin
   if t = 0 then
      if p = 0
         then comment: AC is initially length of input.
            return (1, n)
         else comment: All other processors are idle at time 0.
            return (0, 0)
   S := state(t − 1, p)
   AC := accumulator(S)
   IC := instruction-counter(S)
   case instruction(S) :
      "load l"            : return (IC + 1, local(t − 1, p, l))
      "indirect load l"   : return (IC + 1, local(t − 1, p, local(t − 1, p, l)))
      "global indirect load l"
                          : return (IC + 1, global(t − 1, local(t − 1, p, l)))
      "add", "sub", "read"
                          : similar to "load"
      "store", "fork"     : return (IC + 1, AC)
      "jump l"            : return (l, AC)
      "jump-if-zero l"    : if AC = 0
                               then return (l, AC)
                               else return (IC + 1, AC)
      "halt"              : comment: See if parent activated p in this step.
                            p′ := parent(p)
                            S′ := state(t − 1, p′)
                            if instruction(S′) = "fork l" and
                              local(t − 1, p′, 0) = sibling-count(p)
                               then return (l, accumulator(S′))
                               else comment: p not activated; just pass AC.
                                  return (0, AC)
   end
end
```

Correctness of the simulation is a straightforward induction on t. Implementation of the procedures on an DauxPDA is also easy. Note that each procedure has local variables requiring at most $O(\log n)$ bits of storage, so the DauxPDA needs only that much space on its work tape. The recursion depth is equal to the PRAMs running time, i.e., $O(\log n)$, so the pushdown height will be at most the product of those two quantities, i.e., $O(\log^2 n)$. Each procedure makes at most $O(1)$ recursive calls per recursive level, so the running time of the simulation is $(O(1))^{O(\log n)} = n^{O(1)}$. This completes the proof of Theorem 2. □

The simulation given above is easily adapted to accommodate the generalizations G1-G6 to the definition of CROW-PRAMs proposed earlier. Allowing a more general owner function, say depending on the input or on time (G1,G2) is trivial — just add the appropriate parameters at each call. Using a different processor numbering convention is equally easy, provided that *parent*(p), and *sibling-count*(p) are easily computable (G5). Allowing these functions to be log-space and polynomial time DauxPDA computable will not effect the asymptotic complexity bounds (G6). Bounded multiple ownership (G3), is also easy — in the *global* procedure, where we check whether the owner of global memory cell i wrote into it, we would now need to check among the set of owners to see if any of them wrote. Since this set is only of size $O(1)$, the running time would still be polynomial.

Changing the procedures to accommodate ill-behaved PRAM algorithms (G4) is more subtle. The first change required is that we must now determine the *exact* running time T_a of the algorithm. Using some upper bound $T > T_a$ might cause us to falsely reject due to an invalid global store by some processor *after* T_a. The value of T_a is easily determined by evaluating *state*$(t, 0)$ for $t = 0, 1, \ldots$ until processor 0 halts and accepts. (If it does not accept, there is no need to worry about ownership violations.) The second, and more interesting change, is to check all "store" instructions by all active processors p up to time T_a, basically by doing a depth-first search of the processor activation tree.

procedure *Simulate-G4-CROW-PRAM*
comment: Modified Main Program, incorporating G4.
begin
  $t := 0$
  while $instruction(state(t, 0)) \neq$ "halt" **do** $t := t + 1$
  if $accumulator(state(t, 0)) \neq 1$ **then halt and reject**
  $T_a := t$
  $treewalk(0, 0)$
  halt and accept
end

procedure $treewalk(t, p)$
comment: "Visit" processor p at each time τ between t and T_a, and (recursively) any descendants created during that interval. For each, verify that no non-owner writes occur.
begin
  for $\tau := t$ **to** T_a **do**
    $S := state(\tau, p)$
    if $instruction(S) =$ "global indirect store l" **and**
     $owner(local(t - 1, p, l)) \neq p$
      then halt and reject; **comment:** Owner violation; quit.
    if $instruction(S) =$ "fork"
      then
        $p' :=$ child's processor number
        $treewalk(\tau + 1, p')$
end

Correctness of this procedure is argued as follows. If the CROW-PRAM algorithm has no owner write violations, then the procedure is correct, as before. On the other hand, suppose there is a violation, say at time t by processor p. Our procedures correctly determine the state of the PRAM up until time t. After time t, the state of the PRAM is undefined, whereas our procedure calls return values as if the violation had not occurred. However, eventually *treewalk* will detect the fault. It may reject when evaluating $state(t', p')$ for some $t \leq t' \leq T_a$ with $p' \neq p$, but on a branch of the processor activation tree that happens to be explored *before* p's branch. At the latest, however, it will detect the fault after evaluating $state(t, p)$. We can count on this, since our simulation is faithful up to time $t-1$, and the state of the PRAM at that time contains all the information we need to deduce that processor p is active at time t, and about to execute a "store" in violation of the ownership constraint. Hence, eventually we will evaluate $state(t,p)$, detect the fault, and halt.

The running time of this algorithm is still polynomial, since $treewalk(-,p)$ is called exactly once for each active processor p, and there are at most polynomially many processors to be checked.

Thus we have shown the following.

Theorem 3 *Any set recognized in time $O(\log n)$ on a generalized CROW-PRAM, i.e., one satisfying generalizations G1-G6 of the basic definition, is in LOGDCFL.*

This completes the proof of the "only if" direction of Theorem 1. The converse is shown in the following sections.

4 DPDA Definitions and Notation

We assume familiarity with deterministic pushdown automata (DPDA), as defined for example by Harrison [19], as well as standard variations on this model.

Our DPDAs have state set Q, input alphabet Σ and pushdown alphabet Γ. The empty string is denoted by ϵ, the length of string S by $|S|$, and string concatenation by "$\cdot$". At each step either the current topmost pushdown symbol is popped off the pushdown, or a single new symbol is pushed onto the pushdown above the current symbol. We assume the transition function δ is defined for every possible state, input symbol and pushdown symbol. Thus

$$\delta :\; Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \;\rightarrow\; Q \times (\Gamma \cup \{\mathbf{pop}\})$$

The DPDA begins in state q_0 with $\gamma \in \Gamma$ as the initial pushdown contents, with the input head at the left of the input, and accepts by entering state q_a with γ as the only pushdown contents after having advanced the input head to the right end of the input. We assume the DPDA never fails to read all the input and always empties its pushdown of all symbols except γ at the end of the computation. Furthermore, we assume that for all $\sigma \in \Gamma$ there is some transition pushing σ. By standard techniques (see, e.g., Harrison [19, Section 5.6]), there is a constant $c > 0$ such that the DPDA can be assumed to have the above properties and to halt in time $c \cdot n$ at most, with maximum pushdown depth n, on any input of length n.

The efficient simulation of a DPDA to be described makes use of the concepts of surface configuration and instantaneous description, which are defined relative to a particular input $x = x_0 x_1 \ldots x_{n-1},\ x_i \in \Sigma$. A *surface configuration* is a triple (q, i, σ) where q is a state, i is an integer coded in binary between 0 and n representing the position of the input head, and $\sigma \in \Gamma$ represents the *topmost* pushdown symbol. The set of all surface configurations is denoted U. An *instantaneous description (id)* of the DPDA is a pair $\langle u, S\rangle$ where u is a surface configuration and $S \in \Gamma^*$ is a string representing all but the topmost symbol of the pushdown (with bottommost pushdown symbol represented by the rightmost position of S). For convenience, we refer to S as the *stack*. Thus, the initial id is $\langle (q_0, 0, \gamma), \epsilon\rangle$ and the unique accepting id is $\langle (q_a, n, \gamma), \epsilon\rangle$. An id where the stack component is ϵ is called an *ϵ-id*. (Note an ϵ-id corresponds to a pushdown of one symbol, in the surface configuration.) For an id $I = \langle u, S\rangle$ we define $height(I)$ to be $|S|$, and define projection functions $surface(I) = u$, and $stack(I) = S$.

A surface configuration $u = (q, i, \sigma)$ is said to be *popping* if the transition defined for q, x_i and σ pops the pushdown, and is *pushing* otherwise. An id is popping or pushing as its surface configuration is popping or pushing.

We write $I_1 \vdash I_2$ if id I_2 follows from id I_1 in one step of the DPDA on input x, $I_1 \vdash^t I_2$ if I_2 follows I_1 in exactly t steps, and $I_1 \vdash^* I_2$ if $I_1 \vdash^t I_2$ for some $t \geq 0$.

By our definition, ids only represent configurations of the machine with at least one pushdown symbol; if I_1 is a popping ϵ-id there is *no* id I_2 such that $I_1 \vdash I_2$. Thus, a popping ϵ-id is said to be *blocked*. This is true even though the DPDA makes one final move from I_1 (depending on the input symbol, state, and single pushdown symbol in the surface configuration) to empty its pushdown.

For convenience we assume the final accepting configuration is defined to pop, so that it will be a blocked id. We denote by $\langle u, S_1\rangle \cdot S_2$ the id

$\langle u, S_1 \cdot S_2 \rangle$, i.e., $\langle u, S_1 \rangle$ modified so that the symbols of S_2 are placed below the symbols of S_1 on the stack. We illustrate some of the notation with three useful propositions.

Proposition 4 (*Bottom-padding*) For all surface configurations u, v and strings $S_1, S_2, S_3 \in \Gamma^*$

$$\langle u, S_1 \rangle \vdash^k \langle v, S_2 \rangle \Rightarrow \langle u, S_1 \rangle \cdot S_3 \vdash^k \langle v, S_2 \rangle \cdot S_3.$$

Note the converse is not true in general, but is in the following case.

Proposition 5 (*Bottom-unpadding*) For all surface configurations u, v and strings $S_1, S_2, S_3 \in \Gamma^*$, if

$$\langle u, S_1 \cdot S_3 \rangle \vdash^k \langle v, S_2 \cdot S_3 \rangle$$

and

$$\forall\, j \leq k \;\; \langle u, S_1 \cdot S_3 \rangle \vdash^j I \Rightarrow \mathit{height}(I) \geq |S_3|$$

then

$$\langle u, S_1 \rangle \vdash^k \langle v, S_2 \rangle.$$

Proposition 6 (*Block-continuation*) For all surface configurations u, v, w and strings $S_1, S_2, S_3 \in \Gamma^*$

$$\langle u, S_1 \rangle \vdash^j \langle v, \epsilon \rangle \text{ and } \langle v, S_2 \rangle \vdash^k \langle w, S_3 \rangle \Rightarrow \langle u, S_1 \cdot S_2 \rangle \vdash^{k+j} \langle w, S_3 \rangle.$$

In addition to the restrictions on DPDAs discussed above, we assume that no id can occur twice in a computation of the DPDA when started at any given id [19, Section 5.6]. This justifies using ids as references to particular points in computations. E.g., if $I \vdash^t J$, we could refer to the id J to uniquely identify the point in the computation t steps after id I.

5 The Basic DPDA Simulation Algorithm

We now will describe a procedure to efficiently simulate a DPDA on input x of length n. Our algorithm is motivated by the "repeated doubling" idea used, e.g., by Fortune and Wyllie [15, 39] and Klein and Reif [20], which can be described in our setting as follows.

Suppose we have computed for all surface configurations $u \in U$ and all strings $S \in \Gamma^*$, the 2^k step transition function $D^k(\langle u, S\rangle)$, i.e., $\langle u, S\rangle \vdash^{2^k} D^k(\langle u, S\rangle)$. Then we could easily compute the 2^{k+1} step transition function D^{k+1} by composing D^k with itself:

$$D^{k+1}(\langle u, S\rangle) = D^k(D^k(\langle u, S\rangle)).$$

However, efficiency considerations preclude defining D^k for all possible stacks. Observing that in a computation of 2^k steps only the top 2^k symbols of the stack are accessed, S can be "split" by writing $S = S_1 \cdot S_2$ where S_2 contains everything after the first 2^k symbols of S. (S_2 will be empty if S has length $\leq 2^k$.) Then the above could be rewritten

$$D^{k+1}(\langle u, S_1 \cdot S_2\rangle) = D^k(D^k(\langle u, S_1\rangle) \cdot S_2).$$

Although this could be used to limit the number of stacks considered to those of length at most 2^k, there are still too many for a polynomial number of processors to compute in $O(\log n)$ time. A key observation in constructing an efficient algorithm is that the number of stacks that need to be considered can be much more limited than suggested above. It will be shown that it is sufficient to consider a polynomial-sized set of stacks, provided we use both stack splitting and a somewhat more complicated doubling technique. To simplify the set of stacks considered, we compute a function Δ^k in place of D^k described above, that gives the result after *at least* 2^k steps rather than exactly 2^k steps. The advantage is that we can use appropriately chosen break points to keep the stacks simple.

We first describe the algorithm assuming that all of the stacks are explicitly manipulated. In Section 6, we describe a PRAM implementation that avoids this by using a more succinct representation than the stacks themselves. Two functions on ids are used, Δ^k and LOW^k, each of which is defined inductively on the parameter k. For an id I_1, $\Delta^k(I_1)$ returns an id

I_2 that results after t steps of the DPDA starting in id I_1. The value of t is implicitly determined by the algorithm itself, but it will be shown that $t \geq 2^k$, unless a blocked id is reached from I_1 in less than 2^k steps — in this case t is the number of steps needed to reach the blocked id. Formally, for ids I_1 and I_2, Δ^k will satisfy:

$$\Delta^k(I_1) = I_2 \;\Rightarrow\; I_1 \vdash^t \; I_2 \text{ and either } t \geq 2^k \text{ or } I_2 \text{ is blocked.} \tag{1}$$

The function $LOW^k(I_1)$ returns the id I_2 that is the id of lowest height among all ids in the computation from I_1 to $\Delta^k(I_1)$ inclusive, and if there is more than one id of minimal height in this computation, is the earliest such id, i.e., the one closest to I_1. More formally,

$$\begin{aligned} &LOW^k(I_1) = I_2 \Leftrightarrow \text{ for some } t_1, t_2 \geq 0 : \\ &\quad (a)\; I_1 \vdash^{t_1} I_2 \text{ and } I_2 \vdash^{t_2} \Delta^k(I_1), \\ &\quad (b)\; \forall\; 0 \leq t < t_1,\; I_1 \vdash^t J \;\Rightarrow \mathit{height}(J) > \mathit{height}(I_2), \text{ and} \\ &\quad (c)\; \forall\; 0 < t \leq t_2,\; I_2 \vdash^t J \;\Rightarrow \mathit{height}(J) \geq \mathit{height}(I_2). \end{aligned} \tag{2}$$

Given these definitions, to determine if the DPDA accepts x, it is sufficient to check whether

$$\Delta^{\lceil \log_2 c \cdot n \rceil}(\langle (q_0, 0, \gamma), \epsilon \rangle) = \langle (q_a, n, \gamma), \epsilon \rangle,$$

since the DPDA runs in time at most $c \cdot n$ on any input of length n.

As discussed above it is necessary to restrict the number of stacks on which Δ^k must be defined. By careful definition of Δ the information needed to compute Δ^{k+1} from Δ^k can be restricted to consideration of ids whose stack contents are suffixes of stacks produced by Δ^k operating on ϵ-ids, of which there are only polynomially many, $O(n)$ in fact. To state this more precisely, we define SS^k (mnemonic for "simple stacks") to be the set of strings over Γ^* that represent the bottom portions of stacks in ids in the range of Δ^k operating on all ϵ-ids, i.e.,

$$SS^k = \{S \in \Gamma^* \mid S \text{ is a suffix of } \mathit{stack}(\Delta^k(\langle u, \epsilon \rangle)) \text{ for some } u \in U\}.$$

Because $|U| = O(n)$, SS^k contains $O(n^2)$ elements — one for each $u \in U$ and for each suffix of the unique stack determined by u. To motivate this

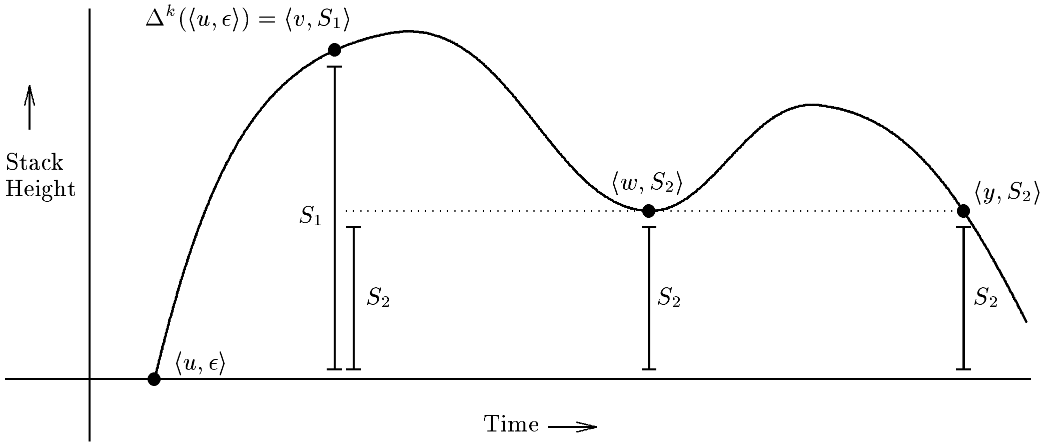

Figure 1: Illustrating SS^k.

definition of SS^k, consider the diagram in Figure 1, plotting stack height versus time in a part of a computation of the DPDA. The diagram shows a stack S_1 built up by a Δ^k-computation starting from $\langle u, \epsilon\rangle$. There must be a complementary computation, starting at $\langle v, S_1\rangle$ that eventually empties this stack. In Figure 1, part of S_1 is removed in the computation starting at $\langle v, S_1\rangle$ and continuing to $\langle w, S_2\rangle$. The rest of S_1 (consisting of S_2) is removed later beginning at $\langle y, S_2\rangle$. Note that S_2 is a suffix of S_1 — which illustrates why SS^k contains not only stacks arising from Δ^k operating on ϵ-ids, but also all suffixes of such stacks.

We will show later that for $k \geq 0$, the stacks in SS^{k+1} are further restricted in that each is the concatenation of two strings in SS^k, i.e.,

$$SS^{k+1} \subseteq SS^k \cdot SS^k \text{ for } k \geq 0. \tag{3}$$

For technical reasons, it will be important to maintain the information specifying how a stack in SS^{k+1} is split into two stacks from SS^k, rather than simply treating stacks as undifferentiated character strings. In the interest of simplicity, however, we will largely ignore this issue in the current section. It will be treated fully in Section 6.

In arguing the correctness of our algorithm, we prove the following by induction on k.

Correctness condition: Δ^k and LOW^k are well-defined for all ids with stacks from SS^k; and Δ^k, LOW^k and SS^k satisfy properties (1), (2) and (3) above, respectively. $(*)$

The crux of our algorithm and its correctness proof is captured by the following lemma, which shows that we can progress at least 2^k steps in the simulation while simultaneously restricting attention to a limited set of stacks, by applying Δ^k only at selected low points.

Lemma 7 (The LOW-Δ Lemma.) *Let $I = \langle u, S\rangle$ be an id with $S \in SS^k$, let $L = LOW^k(I)$, and let $J = \Delta^k(\langle surface(L), \epsilon\rangle) \cdot stack(L)$. Then*

(a) $I \vdash^t J$ for some $t \geq 0$,

(b) if J is unblocked, then $t \geq 2^k$, and

(c) $stack(J) \in SS^k \cdot SS^k$.

Proof: See Figure 2, which plots stack height versus time in the computation of the DPDA. There are three distinct cases. In the first and simplest (not shown in the diagram), the DPDA blocks (attempting to pop when stack height is zero) before completing 2^k steps. In the second, the Δ^k-computation from $\langle surface(L), \epsilon\rangle$ blocks before completing 2^k steps, but we will argue that the overall LOW-Δ computation does complete at least 2^k steps. In the third case, none of the sub-computations block.

Part (a) follows directly from properties (1) and (2).

From correctness property (2), L is the lowest point in the computation from I to L (at least), so $stack(L)$ must be a suffix of $stack(I) = S$, which is in SS^k by assumption. Thus, $stack(L)$ is in SS^k. From the definition of SS^k, $stack(\Delta^k(\langle surface(L), \epsilon\rangle))$ is also in SS^k. Thus, $stack(J)$ is in $SS^k \cdot SS^k$, satisfying (c).

Now assume J is unblocked. Let $M = \Delta^k(\langle surface(L), \epsilon\rangle)$, hence $J = M \cdot stack(L)$. If M is itself unblocked, then from the correctness property (1) for Δ^k, J is at least 2^k steps past L and part (b) follows. On the other hand, if M is blocked but J is unblocked, then $stack(L)$ must have non-zero height. In this case J cannot precede $\Delta^k(I)$, since otherwise the id succeeding J would be a point of lower height than L in the range from I

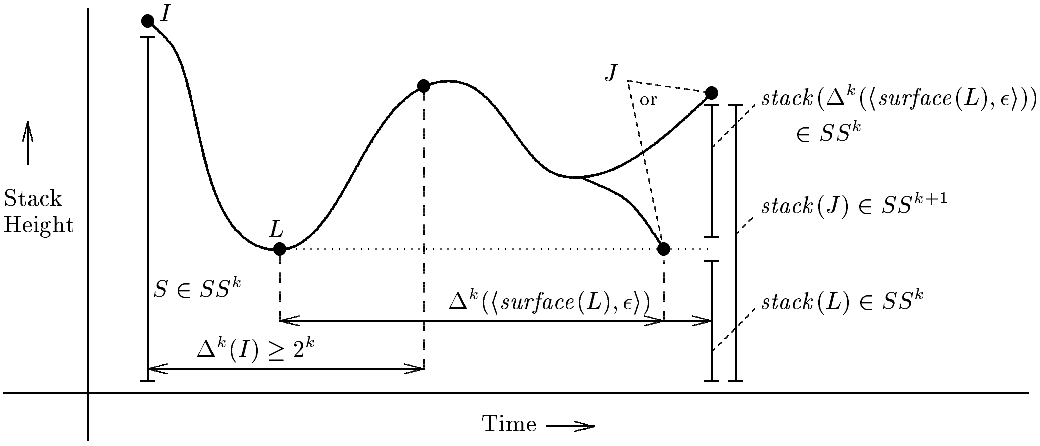

Figure 2: The LOW-Δ Lemma.

to $\Delta^k(I)$, inclusive, contradicting correctness property (2). It follows that $\Delta^k(I)$ is unblocked, and part (b) again follows from correctness property (1).

□

The expression for J in the lemma above occurs so frequently that it is convenient to introduce a special notation for it. We define $\widetilde{\Delta}^k(L)$ to be $\Delta^k(\langle surface(L), \epsilon\rangle) \cdot stack(L)$. For example, the LOW-Δ Lemma shows that $\widetilde{\Delta}^k(LOW^k(I))$ either progresses 2^k steps or blocks.

Note that for I, L, J as in the LOW-Δ Lemma, if $height(L) > 0$, then J is necessarily unblocked, and so $\widetilde{\Delta}^k(LOW^k(I))$ necessarily progresses 2^k steps.

The LOW-Δ Lemma applies to an id I only when its stack is in SS^k. We will need an analogous result when I has a stack consisting of two or three segments each from SS^k. The desired low point in such a stack is found by the following *Iterated LOW* function. It will be useful later to define the function to handle any constant number d of stack segments rather than just three. See Figure 3.

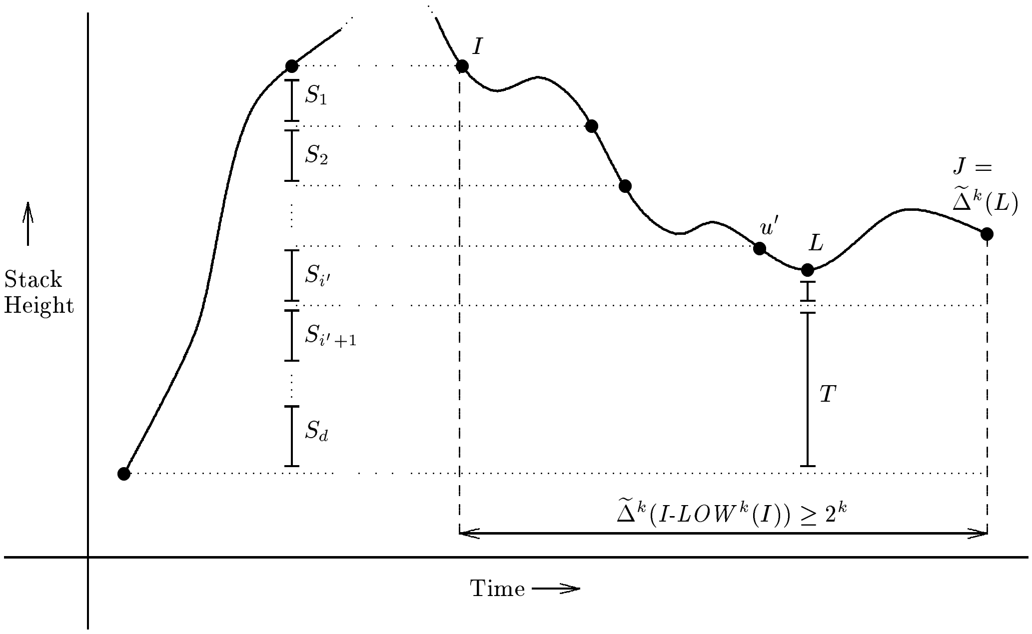

Figure 3: *I-LOW^k*.

function *I-LOW^k*$(I : \text{id})$ **returns** id
comment: Assuming $I \in U \times (SS^k)^d$, return the id of a LOW^k point of nonzero height in a computation from I, if one exists. If not, return the resulting ϵ-id.
begin
  let $I = \langle u, S_1 \cdot S_2 \cdot \cdots \cdot S_d \rangle$, where $S_1, S_2, \ldots, S_d \in SS^k$
  for $i := 1$ **to** d **do**
    $\langle u, S \rangle := LOW^k(\langle u, S_i \rangle)$
    if $\mathit{height}(\langle u, S \rangle) > 0$
      then return $\langle u, S \cdot S_{i+1} \cdot S_{i+2} \cdot \cdots \cdot S_d \rangle$
  comment: Every segment emptied.
  return $\langle u, \epsilon \rangle$
end

The desired generalization of the LOW-Δ Lemma is the following.

Lemma 8 (The *I-LOW-*Δ Lemma.) *Let $I = \langle u, S\rangle$ be an id with $S \in (SS^k)^d$, and let $J = \widetilde{\Delta}^k(I\text{-}LOW^k(I))$. Then*

(a) $I \vdash^t J$ for some $t \geq 0$,

(b) if J is unblocked, then $t \geq 2^k$, and

(c) $stack(J) \in (SS^k)^{d+1}$.

Proof: Part (a) follows from propositions (1) and (2). Let $L = I\text{-}LOW^k(I)$. Since $I\text{-}LOW^k$ modifies the stack of its argument only by calling LOW^k, it follows that $stack(L)$ is a suffix of $stack(I)$, and hence by hypothesis is in $(SS^k)^d$. The stack segment added by the call to $\widetilde{\Delta}^k$ is in SS^k, establishing part (c).

The key point in establishing (b) is that $L = I\text{-}LOW^k(I)$ is a "LOW^k point," hence the LOW-Δ Lemma can be applied. Specifically, let i' be the last value taken by i in the **for** loop, and let u' be the value taken by u before the last call to LOW^k. Let $I' = \langle u', S_{i'}\rangle$, and $L' = LOW^k(I')$. Note that L' is the last value taken by $\langle u, S\rangle$ before return. Then, letting $J' = \widetilde{\Delta}^k(L')$, and $T = S_{i'+1} \cdot \cdots \cdot S_d$, it is easy to see that $I \vdash^* I' \cdot T$, $L = L' \cdot T$, and $J = J' \cdot T$. Now, the LOW-Δ Lemma applies to I', L', J'. In particular, if J' is unblocked, then it is at least 2^k steps past I', hence J is at least 2^k past I, satisfying part (b). Thus it suffices to show that J' is unblocked whenever J is unblocked. There are two cases to consider. First, suppose $I\text{-}LOW^k$ returns because $height(L') > 0$. Then as noted earlier, J' will necessarily be unblocked. On the other hand, if $I\text{-}LOW^k$ returns with $height(L') = 0$, then by inspection $i' = d$, hence $T = \epsilon$, so $J' = J$. Thus in either case J is unblocked if and only if J' is unblocked, and part (b) follows. $\square$

In the code for $I\text{-}LOW^k$ given above we do not indicate how to determine the decomposition of its stack parameter into d segments from SS^k. In brief, as suggested in the remark following the definition of SS^k, we will retain this decomposition information when the stacks are initially computed. Detailed explanation of this issue is deferred to the next section.

In defining LOW^k, it will be convenient to use an auxiliary function "*min*", that takes as argument a sequence of ids and returns the id of minimal height in the sequence. If there are several of minimal height, it returns the leftmost; for our applications, this will always be the earliest in time.

Construction: We are finally ready to define Δ^k and LOW^k for all $k \geq 0$.

[*Correctness:* Following the parts of the definitions of the functions, we provide, enclosed in square brackets, appropriate parts of the correctness arguments establishing that Δ^k, LOW^k, and SS^k satisfy $(*)$.]

Basis ($k = 0$): For all ids I with $height(I) \leq 1$:

$$\Delta^0(I) \;=\; \begin{cases} J & \text{if } \exists\, J \text{ such that } I \vdash J \\ I & \text{otherwise (i.e., if } I \text{ is blocked),} \end{cases}$$

and

$$LOW^0(I) \;=\; min(\, I, \Delta^0(I)\,).$$

[*Correctness:* By our assumption that for all $\sigma \in \Gamma$ there is some state pushing σ, we see that SS^0 must be exactly $\Gamma \cup \{\epsilon\}$, which is exactly the set of stacks in the domain of Δ^0 and LOW^0. By inspection, for all I in this domain $I \vdash^t \Delta^0(I)$, where $t \geq 2^0$ unless $\Delta^0(I)$ is blocked. Thus (1) is satisfied. (2) holds because there are only two points in the range of points under consideration, and min selects the lower of these. (3) holds vacuously.]

The inductive definition of Δ^{k+1} and LOW^{k+1} is done in two phases, first considering ids with empty stacks, which determine SS^k, then considering ids with stacks in $SS^k - \{\epsilon\}$.

Inductive Definition of Δ^{k+1} and LOW^{k+1} on empty stacks: (See Figure 4.) For $k \geq 0$, and for all $u \in U$:

$$\Delta^{k+1}(\langle u, \epsilon\rangle) \;=\; \widetilde{\Delta}^k(LOW^k(\Delta^k(\langle u, \epsilon\rangle))),$$

and

$$LOW^{k+1}(\langle u, \epsilon\rangle) \;=\; \langle u, \epsilon\rangle.$$

Basically, this procedure computes Δ-LOW-Δ. Assuming the computation does not block, the id reached by the first Δ is 2^k steps past the starting point, *and* satisfies the hypothesis of the LOW-Δ Lemma. Thus, the subsequent LOW-Δ pair achieves another 2^k steps progress, *and* keeps the resulting stack simple (i.e., in SS^{k+1}). This argument is the main ingredient in the correctness proof, below. The case where the initial id has a

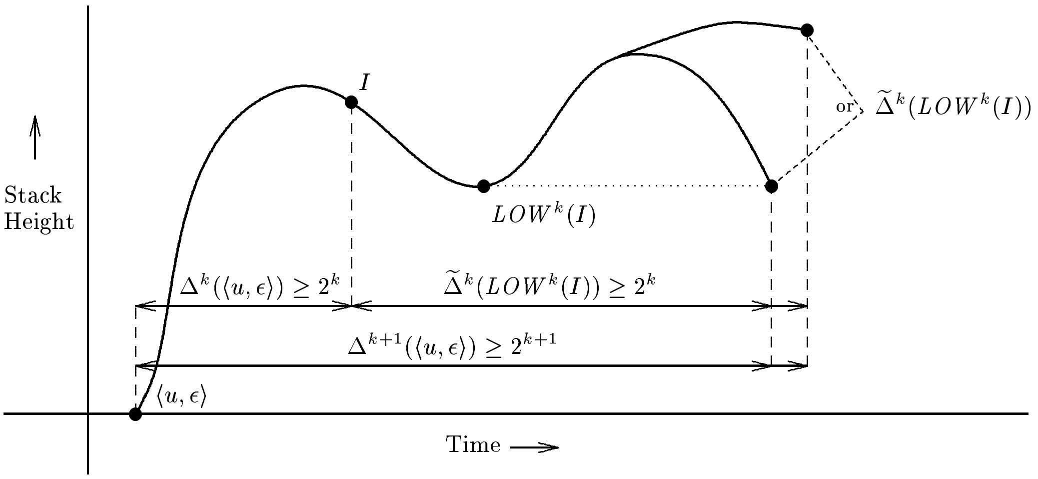

Figure 4: $\Delta^{k+1}(\langle u, \epsilon\rangle)$.

non-empty stack will turn out to be similar, except that we need to precede this with another LOW or two.

[*Correctness:* Let $I = \Delta^k(\langle u, \epsilon\rangle)$. Note that by the definition of SS^k, $stack(I) \in SS^k$, so the hypothesis of the LOW-Δ Lemma is satisfied by I. If $\Delta^{k+1}(\langle u, \epsilon\rangle)$ is blocked, then (1) is immediately satisfied. If it is not blocked, then neither is I, so I is at least 2^k steps past $\langle u, \epsilon\rangle$, by property (1). Applying the LOW-Δ Lemma, $\widetilde{\Delta}^k(LOW^k(I))$ is at least 2^k steps past I, hence 2^{k+1} past $\langle u, \epsilon\rangle$. Thus, $\Delta^{k+1}(\langle u, \epsilon\rangle)$ also satisfies (1). Clearly $\langle u, \epsilon\rangle$ is the earliest id of height zero at or after itself, so property (2) is trivially satisfied by $LOW^{k+1}(\langle u, \epsilon\rangle)$. Property (3) follows directly from the LOW-Δ Lemma.]

To complete the definition of Δ^{k+1} and LOW^{k+1} we must now define them on all ids with non-empty stacks $S \in SS^{k+1}$ (as defined by Δ^{k+1}'s action on ϵ-ids).

Inductive Definition of Δ^{k+1} and LOW^{k+1} on non-empty stacks:

Using *I-*LOW^k we define $\Delta^{k+1}(I)$ and $LOW^{k+1}(I)$ for all $I = \langle u, S\rangle$ with $u \in U$, and $S \in SS^{k+1} - \{\epsilon\}$ as the result of the following computations (see

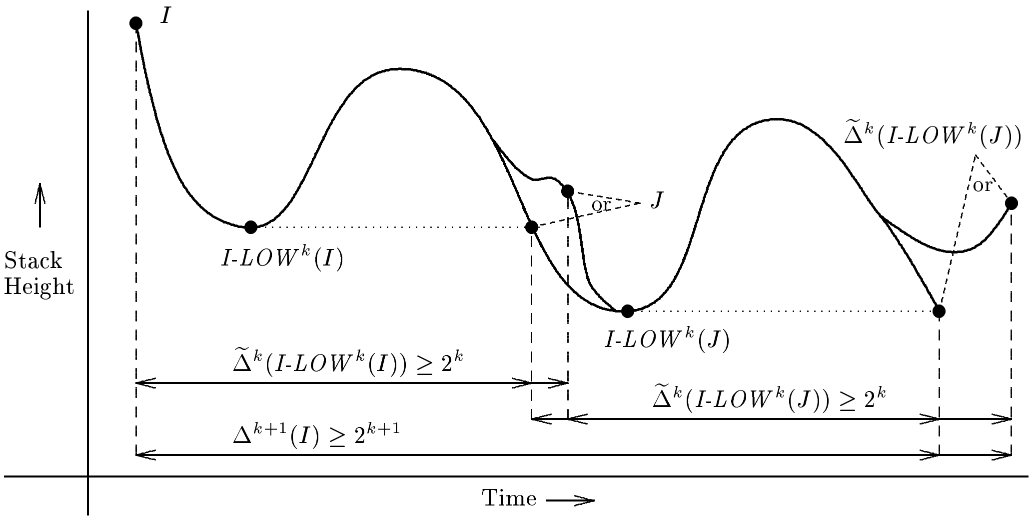

Figure 5: $\Delta^{k+1}(I), stack(I) \neq \epsilon$.

Figure 5):

$$\begin{aligned} J &= \widetilde{\Delta}^k(I\text{-}LOW^k(I)), \\ \Delta^{k+1}(I) &= \widetilde{\Delta}^k(I\text{-}LOW^k(J)), \end{aligned}$$

and

$$LOW^{k+1}(I) = min(I\text{-}LOW^k(I), I\text{-}LOW^k(J)).$$

[*Correctness:* Property (1) follows immediately by applying the *I-LOW*-Δ Lemma twice. Property (2) is satisfied by $LOW^{k+1}(I)$ since the two points to which *min* is applied subsume all the low points of all the subcomputations comprising Δ^{k+1}. Property (3) is inapplicable.]

We remark that from the *I-LOW*-Δ Lemma $stack(J)$ above may consist of three stack segments, even though $stack(I)$ contains only two. This is the main reason for defining $I\text{-}LOW^k$ on more than two stack segments.

Finally, we remark that $I\text{-}LOW^k$ is the identity function on ids with empty stack, and is equal to LOW^k when $d = 1$. Thus, when $stack(I) = \epsilon$, the above definition reduces to exactly the same computation as given earlier for the empty stack case, since $J = \tilde{\Delta}^k(I\text{-}LOW^k(I)) = \Delta^k(I) \in U \times (SS^k)^1$ in this case. Similarly, this definition of $LOW^{k+1}(I)$ also suffices in the case when $stack(I) = \epsilon$. Thus, one could use these more general definitions to handle both cases.

This completes the definitions of Δ and LOW, and the proof of their correctness. To summarize, the key features of this construction are that LOW^{k+1} and Δ^{k+1} each require only a constant number of calls to the level k procedures, they guarantee at least 2^{k+1} progress in the simulation, and they need to be defined on domains of only polynomial size. In the next two sections we will exploit these features to give fast implementations on PRAMs, and small space implementations on PDAs.

6 CROW-PRAM Implementation

The one important issue ignored in the discussion so far is the question of efficiently handling the stacks. To obtain the desired $O(\log n)$ running time, we need to manipulate stacks of length $\Omega(n)$ in unit time. In particular, when defining $\Delta^{k+1}(\langle u, S\rangle)$ and $LOW^{k+1}(\langle u, S\rangle)$, where $S \in SS^{k+1} \subseteq SS^k \cdot SS^k$, it is necessary to be able to split S into two segments, each a stack in SS^k. This can be done by retaining the information splitting S into components when S is originally constructed by $\Delta^{k+1}(\langle v, \epsilon\rangle)$ for some v. In fact, the decomposition information is really the only information about S needed to apply the inductive definitions — the actual contents of the stacks are never consulted in the definitions, except in the base cases. This fact allows us to replace the actual stacks with abbreviations, avoiding the explicit manipulation of long character strings, provided the decomposition information is kept available.

We now introduce the more succinct notation for stacks, revise the algorithms using this notation, and then discuss the CROW-PRAM implementation using this notation.

By definition any stack $S \in SS^k$ is a suffix of $stack(\Delta^k(\langle u, \epsilon\rangle))$ for some surface configuration u. We can name S by specifying k, u, and a value h giving the length of the suffix being considered.

Definition: A *stack reference of level* $k \geq 0$, abbreviated "(k)*-reference*," is a pair (u, h) with $u \in U$ and $0 \leq h$. If $0 \leq h \leq \mathit{height}(\mathit{stack}(\Delta^k(\langle u, \epsilon \rangle)))$, the stack reference is said to be *valid*. A (k)-reference (u, h) is said to have *base* u, *height* h, and *level* k. For convenience, ϵ will also be considered a valid (k)-reference, denoting the empty stack of height 0.

For $k \geq 0$, the algorithm will maintain an array $\mathit{SUMMARY}^k$ indexed by surface configurations. The value stored in $\mathit{SUMMARY}^0[u]$ will be the actual symbols of $\mathit{stack}(\Delta^0(\langle u, \epsilon \rangle))$. The value of $\mathit{SUMMARY}^{k+1}[u]$ will be a pair of valid (k)-references, which will turn out to (recursively) specify the actual symbols of $\mathit{stack}(\Delta^{k+1}(\langle u, \epsilon \rangle))$.

A valid (k)-reference (u, h) may refer to any suffix of $\mathit{stack}(\Delta^k(\langle u, \epsilon \rangle))$. Thus, it is convenient to extend the summary notation to handle references. The summary of a valid (k)-reference $R = (u, h)$, is denoted $\mathit{SUMMARY}^k[R]$. For $k = 0$ it is the length h suffix of $\mathit{SUMMARY}^0[u]$. For $k \geq 1$ it is the pair of (k)-references from $\mathit{SUMMARY}^k[u]$ adjusted to height h. This adjustment is carried out as follows. Suppose $\mathit{SUMMARY}^k[u]$ is the ordered pair of (k)-references (v_1, h_1) and (v_2, h_2). If $h > h_2$, then $\mathit{SUMMARY}^k[R]$ is the ordered pair $(v_1, h - h_2)$ and (v_2, h_2), and otherwise it is the single (k)-reference (v_2, h). This corresponds to popping the referenced stack until the desired height h is reached.

Below, we define variants ∇^k, L^k, I-L^k, MIN of the functions Δ^k, LOW^k, I-LOW^k, *min*, respectively, of Section 5, that will operate using stack references and their summary information in place of the stacks themselves. The function MIN behaves like the version of Section 5, except that it now returns the surface configuration and height (rather than the full id) of the leftmost (earliest) of those of its arguments that are of minimum height. The code for ∇^k only needs to be provided for the case of an empty stack. (The definition of $\Delta^k(\langle u, S \rangle)$, $S \neq \epsilon$ was given in Section 5 only to support the definition of $\mathit{LOW}^k(\langle u, S \rangle)$ and the associated correctness assertions; it was not otherwise used.) Finally, we note that, in the code below, the contents of global array $\mathit{SUMMARY}^k$ are always set by the function ∇^k before being referenced by L^k.

function $\nabla^0(u$: surface) **returns** (surface, (0)-reference)
comment: Returns the surface and reference corresponding to $\Delta^0(\langle u, \epsilon\rangle)$, and as a side effect, stores $stack(\Delta^k(\langle u, \epsilon\rangle))$ in the global array $SUMMARY^0[u]$.
var
  v : surface
  R : (0)-reference
begin
  if u is popping **then**
    $SUMMARY^0[u] := \epsilon$
    return (u, ϵ)
  let $\langle u, \epsilon\rangle \vdash \langle v, \sigma\rangle$ where $\sigma \in \Gamma$;
  $SUMMARY^0[u] := \sigma$
  $R := (u, 1)$
  return (v, R)
end

function $L^0(u$: surface, R: (0)-reference)
  returns (surface, (0)-reference)
comment: Returns the surface and reference of the low point in the interval (u, R) to $\nabla^0(u, R)$.
var
  v : surface
  S, S_1 : string
begin
  if $R = \epsilon$ **then return** (u, ϵ)
  $S := SUMMARY^0[\,R\,]$
  let $\langle u, S\rangle \vdash \langle v,\ S_1\rangle$ **comment:** Either $S_1 = \epsilon$ or $S_1 = \gamma_1\gamma_2, \gamma_i \in \Gamma$.
  if $|S| < |S_1|$ **then return** $(u,\ R)$
  return $(v,\ \epsilon)$
end

function $I\text{-}L^k$(u: surface, $R_1, R_2, \ldots, R_d$: sequence of (k)-references)
  returns (surface, sequence of (k)-references)
comment: Returns the surface and a sequence of (k)-references defining the stack of an unblocked low point (if any) in a computation starting from $(u, R_1 \cdot R_2 \cdot \cdots \cdot R_d)$. Note this procedure handles any fixed number d of (k)-references.
var
  R: (k)-reference
begin
  for $i := 1$ **to** d **do**
    $(u, R) := L^k(u, R_i)$
    if $height(R) > 0$ **then return** $(u,\ R, R_{i+1}, \ldots, R_d)$
  return $(u,\ \epsilon)$
end

function ∇^{k+1}(u: surface) **returns** (surface, $(k+1)$-reference)
comment: Returns the surface and reference corresponding to $\Delta^{k+1}(\langle u, \epsilon\rangle)$ and as a side effect, stores the summary for $stack(\Delta^{k+1}(\langle u, \epsilon\rangle))$ in $SUMMARY^{k+1}[u]$.
var
  $v_2,\ v_3$: surface
  $R_2,\ R_3$: (k)-reference
  R : $(k+1)$-reference
begin
  $(v_2, R_2) := L^k(\nabla^k(u))$
  $(v_3, R_3) := \nabla^k(v_2)$
  $SUMMARY^{k+1}[\,u\,] := (R_3, R_2)$
  $R := (u, height(R_2) + height(R_3))$
  return $(v_3,\ R)$
end

function $L^{k+1}(u:$ surface, $R:$ $(k+1)$-reference$)$
  returns (surface, $(k+1)$-reference)
comment: Returns the surface and reference corresponding to the low point in the interval (u, R) to $\nabla^{k+1}(u, R)$.
var
  $S,\ S_1,\ S_2,\ S_3:$ sequence of (k)-references
  $u_1,\ u_2,\ u_3,\ u',\ w:$ surface
  $R':$ $(k+1)$-reference
  $R_2:$ (k)-reference
  $h,\ h'$: integer
begin
  let R be (w, h)
  $S := SUMMARY^{k+1}[R]$
  $(u_1, S_1) := I\text{-}L^k(u, S)$
  $(u_2, R_2) := \nabla^k(u_1)$
  let S_2 be the result of prepending R_2 to the sequence S_1
  $(u_3, S_3) := I\text{-}L^k(u_2, S_2)$
  $(u', h') := MIN((u_1, S_1),\ (u_3, S_3))$
  $R' := (w, h')$
  return (u', R')
end

Correctness follows from the argument given in Section 5 using the correspondence elucidated below. We inductively define a string $\widehat{R}$ associated with each valid (k)-reference R as follows. With each valid (0)-reference $R = (u, h)$, the string $\widehat{R}$ consists of the length h suffix of the string stored in $SUMMARY^0[u]$. Furthermore, for each $k \geq 0$ and each valid $(k+1)$-reference $R = (u, h)$, we associate (inductively) the string $\widehat{R} = \widehat{R_1} \cdot \widehat{R_2}$, where $SUMMARY^{k+1}[R] = (R_1, R_2)$. By induction on k, one can show that $\widehat{R}$, the string associated with the (k)-reference R returned by $\nabla^k(u)$, is exactly $stack(\Delta^k(\langle u, \epsilon\rangle))$ as defined in Section 5, provided that in the algorithm of Section 5, each stack $\widehat{S}$ is decomposed as specified by $SUMMARY[S]$. (Recall that the exact decomposition used in $I\text{-}LOW^k$ was left unspecified in Section 5. We note that, the proofs given there, in particular the proof of Lemma 8, the I-LOW-Δ Lemma, hold for any decomposition of a stack in $(SS^k)^d$ into d substrings each in SS^k, although we only need the proofs to hold for the specific decomposition given by $SUMMARY$.)

Finally, the functions ∇^k and L^k defined above can be used for a time $O(\log n)$ parallel algorithm for DCFL recognition on a CROW-PRAM. The algorithm tabulates ∇^k, $SUMMARY^k$ and L^k for successively higher values of k.

> **for** $k := 0$ **to** $\lceil \log cn \rceil$ **do**
>   **for all** $u \in U$ **do in parallel**
>   Compute $\nabla^k(u)$ and store in a table in global memory. As a side effect, store $SUMMARY^k[u]$.
>   **for all** $u, v \in U$ and all $h \geq 0$ for which $R = (u, h)$ is a valid (k)-reference **do in parallel**
>   Compute $L^k(v, R)$ and store in a table in global memory.
> **Accept iff** $\nabla^{\lceil \log cn \rceil}((q_0, 0, \gamma)) = ((q_a, n, \gamma), \epsilon)$.

Each iteration of the loop can be performed with a constant number of references to previously stored values of ∇^k, L^k, and $SUMMARY^k$.

The implementation of tables indexed by surface configurations and references, and the initialization of a unique processor for every array entry are done using now-standard parallel RAM programming techniques; see Goldschlager [17] or Wyllie [39] for examples. Each surface and reference can be coded by an integer of $O(\log n)$ bits, which can be used as a table subscript. These techniques also suffice to implement the above algorithm on a CROW-PRAM satisfying restrictions R1–R3.

Since there are only $O(n)$ surfaces and $O(n^2)$ references, the number of array entries (and hence the number of processors) can be kept to $O(n^3)$ by reusing array space rather than having separate arrays for each value of k from 0 to $\log n$. The values of $SUMMARY^k$, for example, can be discarded as soon as the values of $SUMMARY^{k+1}$ have been computed.

Thus we have shown the following theorem.

Theorem 9 *Every DCFL can be recognized by a CROW-PRAM satisfying restrictions R1–R3 in time $O(\log n)$ with $O(n^3)$ processors.*

Theorems 2 and 9 together establish Theorem 1. We also obtain the following corollary.

Corollary 10 *CROW-PRAMs satisfying generalizations G1–G4 can be simulated by CROW-PRAMs subject to restrictions R1–R3 with only a constant factor time loss and with only a polynomial increase in number of processors.*

Proof: It was shown in Section 3 that generalized CROW-PRAMs satisfying G1–G6 can be simulated by deterministic auxiliary PDAs with $\log n$ space and polynomial time, and thus that languages recognized by such machines are in Sudborough's class $LOGDCFL$ of languages log space reducible to deterministic context-free languages. A log space-bounded reduction can be done on a CROW-PRAM in time $O(\log n)$ using the deterministic pointer-jumping technique of Fortune and Wyllie [15]. See Cook and Dymond [8] for a detailed description the simulation of log space by a parallel pointer machine in $O(\log n)$ time, and see Lam and Ruzzo [21] for a simulation of the later model by a $O(\log n)$ time-bounded CROW-PRAM. This simulation is easily made to obey restrictions R1–R3. Finally, by Theorem 9, the resulting language can be recognized by a CROW-PRAM also obeying restrictions R1–R3. □

Following appearance of an earlier version of this paper, Monien, *et al.* [25] gave a CREW-PRAM algorithm for DCFL recognition that, for any $\epsilon > 0$, uses $O(\log n)$ time and $n^{2+\epsilon}$ processors. Their algorithm uses functions similar to ours, and suggests an approach to improving the processor bound of the CROW-PRAM algorithm of Theorem 9.

7 Small Space Sequential Implementation

In Section 3 we presented an algorithm for simulating an $O(\log n)$ time CROW-PRAM by a deterministic auxPDA using polynomial time and $O(\log^2 n)$ stack height. This, combined with Theorem 9, yields an alternate proof of the following result of Rytter.

Theorem 11 *(Rytter [33].) L is accepted by a polynomial time logarithmic space DauxPDA if and only if L is accepted by such a machine that furthermore uses stack height $O(\log^2 n)$.*

An analogous result was previously known for nondeterministic PDAs (Ruzzo [32]), but the best result previous to Rytter's for stack height reduction in DauxPDAs required superpolynomial time (Harju [18]; c.f. [32] for an alternative proof).

Corollary 12 *(Harju [18].) DCFLs are in DauxPDA space* $O(\log n)$ *and stack height* $O(\log^2 n)$.

The following result is also a corollary.

Corollary 13 *(Cook [5]; von Braunmühl, et al. [38].) DCFLs are in* SC^2.

The time bound for the algorithm sketched above, while polynomial, is not particularly attractive. As shown by von Braunmühl, Cook, Mehlhorn, & Verbeek [38], DCFL recognition is in simultaneous space $S(n)$ and time $O(n^{1+\epsilon}/S(n))$ on DTMs with random access to their input tapes, for any $\epsilon > 0$ and any $\log^2 n \leq S(n) \leq n$. Their algorithm makes general use of its space resource, i.e., it is not used as a pushdown store, or even as a stack (in the stack automaton sense; Ginsburg, Greibach, and Harrison [16]).

The goal of the remainder of this section is to sketch an improvement to our algorithm to achieve time bounds matching those of von Braunmühl, *et al.*, while still using a DauxPDA. Our modifications borrow some of the key ideas from the von Braunmühl, *et al.* constructions.

First, we outline a more direct algorithm, bypassing the simulation of a general CROW-PRAM. In Sections 5 and 6, we presented an algorithm for simulating a DPDA, based on the procedures ∇^k and L^k. Our procedure ∇^k sets the global $SUMMARY^k$ array as a side effect, and L^k reads from it. It is easy to reformulate these procedures recursively. In a fully recursive version, ∇^k would return the summary information as an additional component of its function value, and accesses to $SUMMARY^k$ in L^k would be replaced by appropriate calls to ∇^k, to (re-)compute the desired stack summaries.

Recursive procedures have a straightforward implementation on a space-bounded deterministic auxiliary PDA. The auxPDA's work tape needs to be long enough to hold the local variables of a procedure, and the pushdown height must be d times as large, where d is the depth of recursion, to hold d "stack frames", each holding copies of the local variables, return address, etc.

For our procedures, the local variables consist of a few integers plus a bounded number of surfaces, requiring $O(\log n)$ space. The recursion depth is at most $\lceil \log_2 cn \rceil$. Thus, our procedures can be implemented on a DauxPDA using space $O(\log n)$ and pushdown height $O(\log^2 n)$. Furthermore, for our procedures, each level $k+1$ procedure makes a bounded number of calls on level k procedures. Since the depth of recursion is $O(\log n)$, the total number of calls is at most $(O(1))^{O(\log n)} = n^{O(1)}$. Exclusive of recursive calls, each procedure takes time $O(\log n)$ to manipulate the surfaces, etc., plus, if necessary $O(n)$ to read inputs. Thus, the total time for the algorithm is polynomial.

The main idea in improving the time bound is to generalize the construction in Section 5 to give, for any integer $d \geq 2$, procedures ∇_d^k, etc., that reflect computations of length at least d^k, rather than 2^k as before. This is easily done with the machinery we have already developed. For example, ∇_d^k is basically the d-fold composition of $\widetilde{\nabla}_d^k(I\text{-}L_d^k(\cdot))$ with itself. Each level $k+1$ procedure makes $O(d)$ calls on level k procedures. Thus, the number of recursive calls, which is the main component of the running time, will be

$$(O(d))^{\log_d n} = n^{\log_d O(d)} = n^{1+O(1/\log d)}.$$

Again, to keep the induction simple, we can arrange that the stacks that need to be considered are suffixes of those built by $\nabla_d^{k+1}(u)$, which turn out to be the concatenation of suffixes of at most d stacks built by $\nabla_d^k(v)$, for various v's. As before, it is important that the list of these v's provides a succinct but useful "summary" of the stack contents.

One final refinement of this idea is to simulate $S(n)$ steps of the DPDA in the base case of our procedures, rather than just one step. Then ∇_d^k will simulate at least $S(n) \cdot d^k$ steps.

Implementation of these procedures on a DauxPDA with $O(\log n)$ work tape and $O(\log^2 n)$ stack height is straightforward, as before.

Random access to the input tape is useful in our algorithm and in von Braunmühl, *et al.*'s for the following reason. Simulation of pop moves requires recomputation of portions of the stack, necessitating access to the portions of the input read during the corresponding push moves. With ordinary sequential access to the input tape, even though repositioning the tape head may be time-consuming ($\Omega(n)$), von Braunmühl, *et al.* show that DCFL recognition is possible in simultaneous space $S(n)$ and time $O(n^2/S(n))$, for $\log^2 n \leq S(n) \leq n$. This is provably optimal. Our techniques appear likely to be useful in this case as well, although we have not pursued this.

Acknowledgements

We thank Michael Bertol, Philippe Derome, Faith Fich, Klaus-Jörn Lange, Prabhakar Ragde, and Marc Snir for careful reading of early drafts, and for useful discussions. Special acknowledgement is due Allan Borodin, without whom we would never have begun this research.

References

[1] R. J. Anderson and L. Snyder. A comparison of shared and nonshared memory models of parallel computation. *Proceedings of the IEEE*, 79(4):480–487, Apr. 1991.

[2] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[3] A. K. Chandra and M. Tompa. The complexity of short two-person games. *Discrete Applied Mathematics*, 29(1):21–33, Nov. 1990.

[4] S. A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *Journal of the ACM*, 18(1):4–18, Jan. 1971.

[5] S. A. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *Conference Record of the Eleventh Annual ACM Symposium on Theory of Computing*, pages 338–345, Atlanta, GA, Apr.-May 1979. See also [38].

[6] S. A. Cook. Towards a complexity theory of synchronous parallel computation. *L'Enseignement Mathématique*, XXVII(1–2):99–124, Jan.-June 1981. Also in [24, pages 75–100].

[7] S. A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, Feb. 1986.

[8] S. A. Cook and P. W. Dymond. Parallel pointer machines. *Computational Complexity*, 3(1):19–30, 1993.

[9] P. W. Dymond. Indirect addressing and the time relationships of some models of sequential computation. *Int. J. of Computers and Math. with Applications*, 5:195–209, 1979.

[10] P. W. Dymond and S. A. Cook. Hardware complexity and parallel computation. In *21st Annual Symposium on Foundations of Computer Science*, pages 360–372, Syracuse, NY, Oct. 1980. IEEE.

[11] P. W. Dymond, F. E. Fich, N. Nishimura, P. Ragde, and W. L. Ruzzo. Pointers versus arithmetic in PRAMs. *Journal of Computer and System Sciences*, 53(2):218–232, Oct. 1996.

[12] P. W. Dymond and W. L. Ruzzo. Parallel random access machines with owned global memory and deterministic context-free language recognition. In L. Kott, editor, *Automata, Languages, and Programming: 13th International Colloquium*, volume 226 of *Lecture Notes in Computer Science*, pages 95–104, Rennes, France, July 1986. Springer-Verlag.

[13] F. E. Fich. The complexity of computation on the parallel random access machine. In J. H. Reif, editor, *Synthesis of Parallel Algorithms*, chapter 20, pages 843–899. Morgan Kaufmann, 1993.

[14] F. E. Fich and A. Wigderson. Towards understanding exclusive read. *SIAM Journal on Computing*, 19(4):717–727, 1990.

[15] S. Fortune and J. C. Wyllie. Parallelism in random access machines. In *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, San Diego, CA, May 1978.

[16] S. Ginsburg, S. A. Greibach, and M. A. Harrison. Stack automata and compiling. *Journal of the ACM*, 14(1):172–201, 1967.

[17] L. M. Goldschlager. A universal interconnection pattern for parallel computers. *Journal of the ACM*, 29(4):1073–1086, Oct. 1982.

[18] T. Harju. A simulation result for the auxiliary pushdown automata. *Journal of Computer and System Sciences*, 19:119–132, 1979.

[19] M. A. Harrison. *Introduction to Formal Language Theory.* Addison Wesley, 1979.

[20] P. N. Klein and J. H. Reif. Parallel time $O(\log n)$ acceptance of deterministic CFLs on an exclusive-write P-RAM. *SIAM Journal on Computing*, 17(3):463–485, June 1988.

[21] T. W. Lam and W. L. Ruzzo. The power of parallel pointer manipulation. In *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, pages 92–102, Santa Fe, NM, June 1989.

[22] K.-J. Lange and R. Niedermeier. Data-independences of parallel random access machines. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, Thirteenth Conference*, Lecture Notes in Computer Science, pages 104–113, Bombay, India, Dec. 1993. Springer-Verlag.

[23] D. Lin, X. Deng, and P. W. Dymond. Implementing Cole's parallel mergesort algorithm on owner-write parallel random access machines. Technical report, York University Department of Computer Science, 1995.

[24] *Logic and Algorithmic,* An International Symposium Held in Honor of Ernst Specker, Zürich, Feb. 5–11, 1980. Monographie No. 30 de L'Enseignement Mathématique, Université de Genève, 1982.

[25] B. Monien, W. Rytter, and H. Schapers. Fast recognition of deterministic CFL's with a smaller number of processors. *Theoretical Computer Science*, 116(2):421–429, 16 Aug. 1993. Corrigendum, *ibid.*, 123(2):427, 31 Jan. 1994.

[26] R. Niedermeier and P. Rossmanith. On optimal OROW-PRAM algorithms for computing recursively defined functions. *Parallel Processing Letters*, 5(2):299–309, June 1995.

[27] R. Niedermeier and P. Rossmanith. PRAM's towards realistic parallelism: BRAM's. In H. Reichel, editor, *Fundamentals of Computation Theory: 10th International Conference, FCT '95*, volume 965 of *Lecture Notes in Computer Science*, pages 363–373, Dresden, Germany, Aug. 1995. Springer-Verlag.

[28] N. Nisan. CREW PRAMs and decision trees. *SIAM Journal on Computing*, 20(6):999–1007, Dec. 1991.

[29] N. Nishimura. Restricted CRCW PRAMs. *Theoretical Computer Science*, 123(2):415–426, 31 Jan. 1994.

[30] V. R. Pratt and L. J. Stockmeyer. A characterization of the power of vector machines. *Journal of Computer and System Sciences*, 12(2):198–221, Apr. 1976.

[31] P. Rossmanith. The owner concept for PRAMs. In C. Choffrut and M. Jantzen, editors, *STACS 91: 8th Annual Symposium on Theoretical Aspects of Computer Science*, volume 480 of *Lecture Notes in Computer Science*, pages 172–183, Hamburg, Germany, Feb. 1991. Springer-Verlag.

[32] W. L. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences*, 21(2):218–235, Oct. 1980.

[33] W. Rytter. On the recognition of context-free languages. In A. Skowron, editor, *Computation Theory: Fifth Symposium*, volume 208 of *Lecture Notes in Computer Science*, pages 318–325, Zaborów, Poland, Dec. 1984 (published 1985). Springer-Verlag.

[34] W. Rytter. Parallel time $O(\log n)$ recognition of unambiguous context-free languages. *Information and Computation*, 73(1):75–86, 1987.

[35] L. J. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal on Computing*, 13(2):409–422, May 1984.

[36] I. H. Sudborough. On the tape complexity of deterministic context-free languages. *Journal of the ACM*, 25(3):405–414, 1978.

[37] U. Vishkin. Synchronous parallel computation - a survey. Technical Report TR-71, Department of Computer Science, Courant Institute NYU, 1983.

[38] B. von Braunmühl, S. A. Cook, K. Mehlhorn, and R. Verbeek. The recognition of deterministic CFL's in small time and space. *Information and Control*, 56(1-2):34–51, Jan./Feb. 1983.

[39] J. C. Wyllie. *The Complexity of Parallel Computations*. PhD thesis, Cornell University, Department of Computer Science, 1979. TR 79-387.