Almost Optimal Sublinear Time Parallel Recognition Algorithms for Three Subclasses of Context Free Languages

Lawrence L. Larmore *

Wojciech Rytter †

Abstract

Sublinear time almost optimal algorithms for the recognition problem for three basic subclasses of context-free languages (unambiguous, deterministic and linear) are presented. Optimality is measured with respect to the work of the best known sequential algorithm for a given problem.

1 Introduction

The basic aim of parallelism is to reduce the time, however this is frequently done by increasing the total work of the algorithm. By *work* we mean the time-processor product. Polylogarithmic time algorithms typically have much greater work than corresponding sequential algorithms. Sublinear time algorithms are usually better in this sense.

We say that a parallel algorithm for a given problem is α-*optimal* if its work matches the work of the best known sequential algorithm within a factor of $O(n^{\alpha} \log^k n)$, for k constant. For each of three subclasses of context-free languages (CFL for short) considered in this paper, there are polylogarithmic time recognition algorithms which are optimal within a factor of $O(n \log^2 n)$.

We gain efficiency at the expense of parallel time and we present sublinear time recognition algorithms which are α-optimal for α arbitrarily close to 0. Such algorithms cannot be simply

*Department of Computer Science, University of Nevada, Las Vegas, NV 89154–4019, USA. Partially supported by National Science Foundation grants CCR-9112067 and CCR-9503441, and by the University of Bonn. Email:larmore@cs.unlv.edu

†Institute of Informatics, Warsaw University, 02–097 Warszawa. Partially supported by the grant KBN 8T11C01208. Email:rytter@mimuw.edu.pl

derived by naively slowing down the known polylogarithmic time algorithms, since the work would not be decreased. The best known sequential times for the recognition problem for unambiguous, deterministic and linear CFL's are, respectively, $T_U = O(n^2)$, $T_D = O(n)$ and $T_L = O(n^2)$. The main result of this paper is the following theorem.

Theorem 1.1 (Main theorem) *For each of the three subclasses (unambiguous, deterministic, linear) of CFL's, and for any $0 < \alpha < 1$, there is an α-optimal parallel algorithm working in time $O(n^{1-\alpha} \log^2 n)$.*

We remark that the polylog factor is irrelevant, in some sense, since it is asymptotically overwhelmed by even the slightest change in the value of α. Theorem 1.1 is still valid if we remove the $log^2(n)$ factor, however the formulation above will be more convenient to deal with later.

The algorithms presented in this paper mimic the sequential algorithms, but they advance in larger steps. The size of one "large step" is $O(n^\alpha)$. Each large step is performed in parallel in $O(\log n)$ time, and there are $O(n^{1-\alpha})$ such steps which are executed consecutively. The larger α, the faster the algorithm.

Throughout this paper, we use the CREW PRAM model of parallel computation (see for example [5]).

2 Parallel recognition of unambiguous context-free languages

We use a version of the algorithm presented in [8] for parallel computation of some dynamic programming recurrences. Assume $G = (V_N, V_T, P, Z)$ is an unambiguous context-free grammar, where V_N, V_T are the sets of nonterminal and terminal symbols, respectively, P is the set of productions, and Z is a start symbol of the grammar. Without loss of generality, G is in Chomsky normal form, and there are no useless symbols (see [6]).

Assume we are given an input string $w = a_1 a_2 \ldots a_n$. Denote by $w[i,j]$ the substring $a_{i+1} \ldots a_j$. The *recognition problem* is to determine whether w is generated by G.

We explain the main ideas using the algebraic framework of *composition systems* [3]. The composition system corresponding to a given grammar G and the input string w is a triple $S = (N, \otimes, \textsc{init})$, where $N = \{ (A,i,j) : A \in V_N \text{ and } 0 \le i < j \le n \}$

The elements of N are called *items*. Each item (A,i,j) corresponds to the possibility that $w[i,j]$ is derived from the nonterminal A. Let "$\Rightarrow$" be the relation "derives in one step" according to a given grammar and let "$\Rightarrow^+$" be the transitive closure of this relation. We say that an item (A,i,j) is *valid* if $A \Rightarrow^+ w[i,j]$. The set $\textsc{init}$ is a set of "atomic" valid items and the operation "$\otimes$," which we call *composition*, generates larger items from smaller ones, and $x \otimes y \subseteq N$ for all $x, y \in N$. $\textsc{init} \subseteq N$ is the set of initial elements (generators) of the form $(A, i, i+1)$, where $A \Rightarrow a_{i+1}$, and composition is defined as follows:

$$\begin{aligned}(B,i,k)\otimes(C,k,j) &= \{(A,i,j)\ :\ A\in V_N \text{ and } A\Rightarrow BC\}\\ (B,i,k)\otimes(C,k',j) &= \emptyset \text{ if } k\neq k'\end{aligned}$$

For two sets X, Y of items define:

$$X\otimes Y \;=\; \bigcup_{x\in X, y\in Y} x\otimes y$$

For $x=(A,i,j)$ define the *size* of x (written $|x|$) to be $j-i$. If X is a set, we use the notation $\#X$ for the cardinality of X.

Let $h=n^{\alpha}$, where $0<\alpha<1$. We partition the set N into the disjoint subsets, for $0\le k<n/h$

$$N_k \;=\; \{x\in N\ :\ kh<|x|\le(k+1)h\}$$

For $0\le k<n/h$ define the k^{th} *strip* to be $S_k=\{x\in N_k\ :\ x \text{ is valid}\}$.

Fact 2.1 *$\#S_k=O(n^{1+\alpha})$ for each $0<k<n/h$.*

For a set $X\subseteq N$ denote by *Closure*(X) the closure of X with respect to the operation $\otimes$.

Fact 2.2 *The item x is valid if and only if $x\in$ Closure*(INIT).

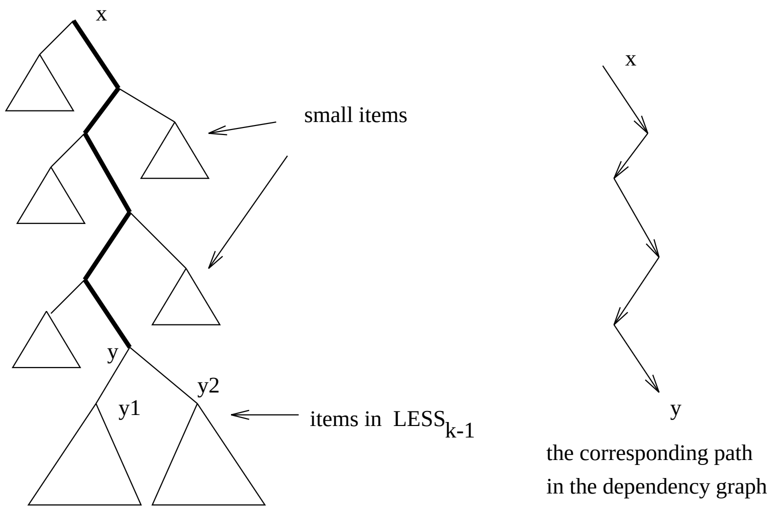

Figure 1: The structure of a generation: $x\in$ *FastClosure*$(\{y\})$. The *small items* are elements of S_0.

Denote, for $X\subseteq S_k$, by *FastClosure*$_k(X)$, the set of all elements in N_k which can be generated from elements in X by multiplying (on the left or right) by elements of S_0. More formally *FastClosure*$_k(X)$ is the smallest subset $Y\subseteq N_k$ such that $X\subseteq Y$ and $(Y\otimes S_0\cup S_0\otimes Y)\cap N_k\subseteq Y$.

Denote by LESS$_k$ the union of all strips with indices smaller than k, for $k>0$. Our algorithm is based on the following easy lemma.

Lemma 2.3 (Correctness lemma) *Assume $k > 0$. Then* $S_k = FastClosure_k(Square_{k-1} \cap N_k)$, *where* $Square_{k-1} = \text{LESS}_k \otimes \text{LESS}_k$.

Proof. Let T be a tree generating $x \in S_k$ from the generators. Let y be a lowest element in T which is in S_k and y_1, y_2 be the sons of y (see Figure 1) Then $y_1, y_2 \in \text{LESS}_k$ and there is a path from x to y in G. Since $y_1, y_2 \in \text{LESS}_k$, we have $y \in \text{LESS}_k \otimes \text{LESS}_k$. All elements "hanging" from the main branch are small items, *i.e.*, elements of S_0. Hence $x \in FastClosure_k(\text{LESS}_k \otimes \text{LESS}_k \cap N_k)$. ∎

The correctness of the algorithm below follows from Lemma 2.3. 'The computation of $Square_k$ is done (according with formula given in Lemma 2.3) in two steps 2.1, 2.2, for implementation reasons

Algorithm Compute_Closure ;

1: {preprocessing} compute the strip S_0; $Square_0 := \text{LESS}_1 := S_0$;

2: **for** $k = 1$ **to** $n^{1-\alpha}$ **do**

begin {main iteration}

comment: $Square_{k-1} = \text{LESS}_k \otimes \text{LESS}_k$.

2.1: $S_k := FastClosure_k(Square_{k-1} \cap N_k)$;

2.2: $New_k := \text{LESS}_k \otimes S_k \cup S_k \otimes \text{LESS}_k \cup S_k \otimes S_k$;

2.3: $Square_k := Square_{k-1} \cup New_k$;

2.4: $\text{LESS}_{k+1} := \text{LESS}_k \cup S_k$

end {main iteration }

3: **return** $\bigcup_k S_k$.

Lemma 2.4 *The total work of all iterations needed to perform Step 2.2 of the algorithm (computing all New_k) is $O(n^2 \log n)$.*

Proof. The following claim is easily derived from the unambiguity of the grammar.

Claim 1. The sets New_k computed in the algorithm are pairwise disjoint.

We need a data structure to perform the operation $X \otimes Y$ with the work proportional to the size of the result. We assume the following list representation of the set $X \subseteq N$. For each position k and nonterminal A, we keep (as a list) the sets

$$\begin{aligned} \text{LEFT}_X(k, A) &= \{i \,:\, (A, i, k) \in X\} \\ \text{RIGHT}_X(A, i) &= \{j \,:\, (A, k, j) \in X\} \end{aligned}$$

The computation of $X \otimes Y$ involves processing all productions $A \to BC$ and all positions k.

$$X \otimes Y \;=\; \bigcup_{k,\, A \to BC} \{(A, i, j) \,:\, i \in \text{LEFT}_X(k, A) \text{ and } j \in \text{RIGHT}_Y(C, k)\}$$

The operations on the lists can be done in logarithmic time. This shows:

Claim 2. Assume that X, Y are sets of valid items. Then $X \otimes Y$ can be computed in logarithmic time, with work proportional to the size of the result.

The work done during each iteration is proportional to the number of newly generated elements. The newly generated sets are pairwise disjoint (due to Claim 1) and their total size is quadratic. Hence the total work of the algorithm is also quadratic. ∎

Let $G_k = (N_k, E_k)$, called the *dependency graph*, where the set of edges is

$$E_k = \{(x,y) \ : \ x = y \otimes z \text{ or } x = z \otimes y \text{ for some } z \in S_0\}$$

An example of a dependency path in the graph G_k is illustrated in Figure 1.

Fact 2.5 $\#E_k = O(n^{1+2\alpha})$ *and* $x \in \mathit{FastClosure}_k(X)$ *if and only if there is a path in* G_k *from* x *to some* $y \in X$.

Lemma 2.6

1. *The computation of Step 1 can be done in* $O(\log^2 n)$ *time with* $n^{1+2\alpha}$ *processors.*
2. *Assume* $X \subseteq S_k$ *and the set* S_0 *is precomputed. Then* $\mathit{FastClosure}_k X$ *can be performed in* $O(\log n)$ *time with* $n^{1+2\alpha}$ *processors.*

Proof.
Point (1).
We refer the reader to [4], where it was shown that the closure of set of initial items (elements of INIT) for an unambiguous grammar can be found in $O(\log^2 n)$ time, with the number of processors proportional to the number of edges of the dependency graph, which is $O(n^{1+2\alpha})$.

Point (2).
According to [4] the set of vertices from which a node of X is reachable can be computed in $O(\log n)$ time, with the number of processors proportional to the number of edges, using a version of parallel tree contraction and a special property of the graph G_k, namely uniqueness of paths from one vertex to another [4]. ∎

Lemma 2.6 implies that Step 2.1 can be performed in $O(\log n)$ time with $n^{1+2\alpha}$ processors.

The two preceding lemmas directly imply the main result of this section:

Theorem 2.7 *Assume an unambiguous CFL is given by a context-free grammar. Then, for an input string* w *of length* n*, we can check if* w *is generated by the grammar in* $O(n^{1-\alpha}\log n)$ *time with* $O(n^{2+\alpha})$ *work, for any* $0 < \alpha < 1$.

3 Parallel Recognition of Deterministic Context-Free Languages

In this section we show how to simulate deterministic pushdown automata in parallel with total work close to linear. Let $h = n^{\alpha}$. Our simulation will take $O(n^{1-\alpha})$ "large" steps sequentially. Each large step, working in logarithmic parallel time, advances the computation by at least h, except perhaps the last step. Thus, the simulation is a combination of a sequential and parallel computation.

We refer the reader to [1] or [7] for the definition of a *one-way deterministic pushdown automaton* (DPDA for short). Let A be a DPDA. Assume that we are given an input string w of length n. We can assume that at each step the height of the stack changes by 1 or -1. A one-step computation which increases the height of the stack we call an UP *move*, while a one-step computation which decreases the height of the stack we call a DOWN *move*. A *surface configuration* (simply *configuration* for short) is the description of the information accessible to the control of the DPDA plus the position of the input head at a given moment. Formally, the surface configuration is a triple:

$$x = \langle \textit{state},\ \textit{position},\ \textit{top symbol of stack} \rangle$$

We distinguish two types of configurations. A configuration x is a *pop* configuration if A makes a pop move while it is configuration x. Otherwise, x is a *push* configuration.

We define a *partial configuration* to be a pair $\langle \textit{state},\ \textit{position} \rangle$, and we define a *total configuration* to be a pair $\langle x, \alpha \rangle$ where x is a configuration and α is the contents of the stack. Of course, the top symbol of α must be compatible with x.

We define a *subcomputation* to be a pair $\langle x, y \rangle$ of configurations such that A will eventually reach y with a one element stack after it starts at x with a one element stack (see Figure 2). We define the *size* of that subcomputation, denoted $|\langle x, y \rangle|$, to be the absolute difference of the positions of x and y, *i.e.*, the number of input symbols read between x and y. In the worst case, there are quadratically many subcomputations. However, it suffices to consider a subset of linear size, as shown below.

Define a configuration y to be a *successor* of x if $\langle x, y \rangle$ is the shortest (smallest length) subcomputation starting at x. In this case we say y is the *successor* of x, and write $y = Succ(x)$, (see Figure 6). Observe that there is only a linear number of the pairs $\langle x, Succ(x) \rangle$.

Fix a parameter $0 < \alpha < 1$. If y is the successor of x and $|\langle x, y \rangle| \leq n^{\alpha}$, we say that the subcomputation $\langle x, y \rangle$ is *small*. A subcomputation $\langle x, y \rangle$ is said to be a *shortcut* if it can be decomposed into a sequence of zero or more small subcomputations and is maximal with respect to this property. Note that $\langle x, x \rangle$ is a shortcut if x is not the beginning of any small computation.

Lemma 3.1
(1) *The set of all small subcomputations can be computed in $O(n^{\alpha})$ time with n processors or in $O(\log^2 n)$ time with $n^{1+\alpha}$ processors.* (2) *If the set of all small subcomputations is computed then the set of all shortcuts can be computed in $O(\log n)$ time with n processors.*

Proof. (1) We assign one processor to each configuration x. The assigned processors sequentially

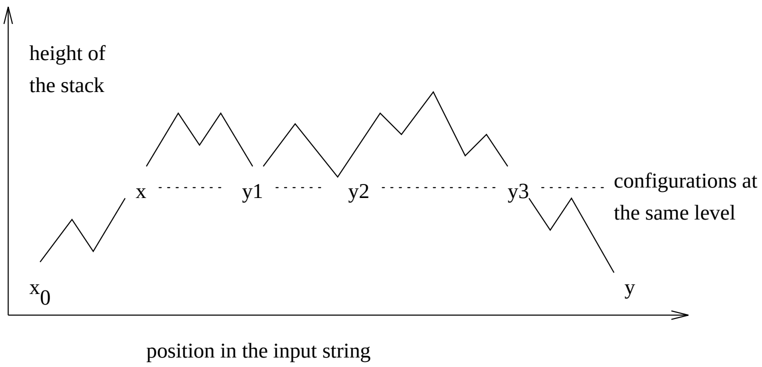

Figure 2: The history of a computation of a DPDA as a "mountain range." We have that $(x, y_1), (x, y_2), (x, y_3)$ are subcomputations. $y_1 = Succ(x)$, $y_2 = Succ(y_1)$ and $y_3 = Succ(y_2)$.

simulate n^α steps of the DPDA. If there are n processors, this requires $O(n^\alpha)$ time. An algorithm which works in $O(\log^2 n)$ time using $n^{1+\alpha}$ processors can be constructed as a version of an algorithm in [4].

(2) The shortcuts can be computed by iterating the operator *Succ* logarithmically many times. For each configuration x, one processor suffices. ∎

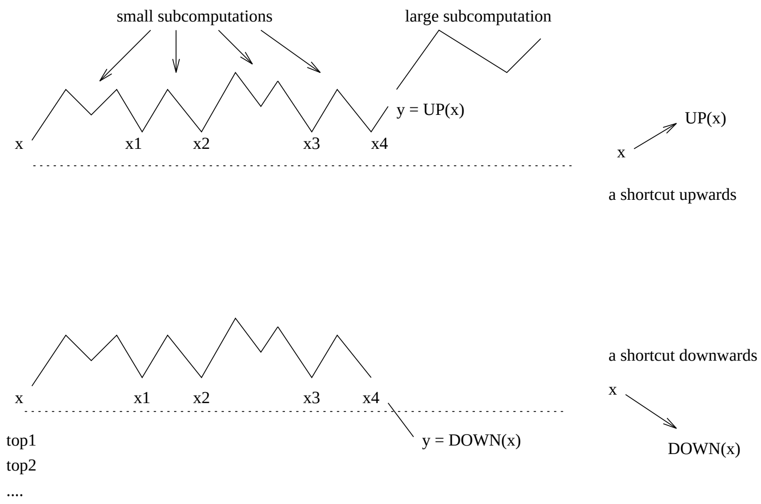

Figure 3: The functions DOWN and UP correspond to maximal chains of small computations on the same level, followed by a pop or a push move, respectively. In the case of a downward shortcut y is the *partial* configuration, the top of the stack is not specified in y, the actual top symbol $top1$ is the same as the one which was immediately below x.

Assume that $\langle x, z\rangle$ is a shortcut. If z is a push configuration then the configuration immediately following z is denoted by $\text{UP}(x)$ (see Figure 3, where $z = x_4$). If z is a pop configuration, then the partial configuration following z is denoted by $\text{DOWN}(x)$ (see Figure 3). Observe that $\text{DOWN}(x)$

does not determine completely the next configuration, since this depends on the top symbol which is below x So if we have a partial configuration $x_1 = \text{DOWN}(x)$, the actual top stack symbol is top_1, and we apply next (for example) a DOWN move, then the next configuration is $x_2 = \text{DOWN}(x_1, top_1)$ (see Figure 4).

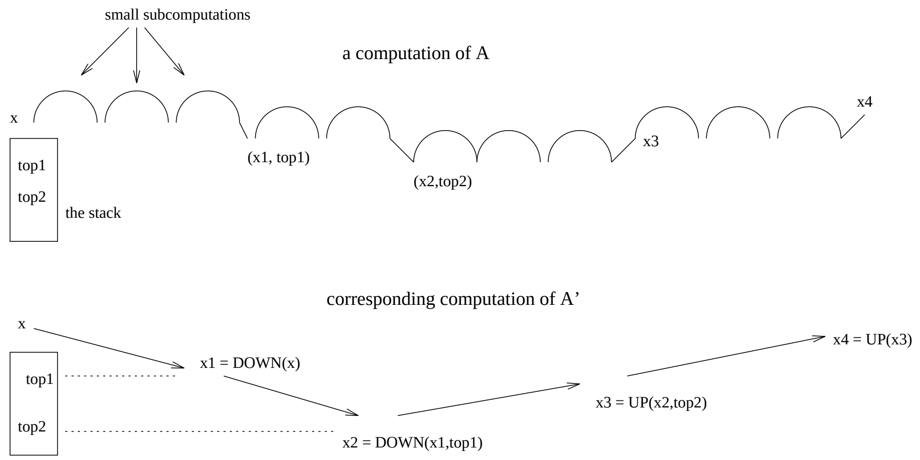

Figure 4: The generalized automaton A' is an accelerated version of A. Observe that x_1, x_2 are partial configurations. The top stack symbols are provided by the stack at the beginning of the computation.

For a given DPDA A and input word w, define a *generalized* DPDA A' which speeds up A by using shortcuts. Each move of A' corresponds to a single UP or DOWN operation (see Figure 4). A' can shift its input head by any number of positions in one move, according to its transition table, which consists of the transition table of A and the information about all shortcuts of A on w. A' works as follows. Assume that A' is in the configuration x and $\langle x, x' \rangle$ is a shortcut. If x' is a push configuration of A then the next configuration of A' is $\text{UP}(x)$ and A' pushes onto the stack the same symbol as A when it moves from x'. If x' is a pop configuration, then A' pops the stack and the next partial configuration of A' is $\text{DOWN}(x)$.

The configurations in which A' makes a pop move are called DOWN *configurations* and the configurations in which A' reduces the stack are called UP *configurations*.

Observe that a DOWN configuration in A' is not necessarily a pop configuration in A.

We say that a sequence of consecutive moves of A' is *one-turn* if it consists of a sequence of DOWN moves followed by a sequence of UP moves, followed by a DOWN move if the last configuration is a DOWN configuration. So it is the maximal sequence of a type:

$\text{DOWN}^* \cdot \text{UP}^* \cdot (\text{DOWN} \vee \epsilon)$, where ϵ is the empty sequence.

Lemma 3.2 *Assume A starts with some stack σ and a configuration x. Assume A makes at least h steps. Let z be a configuration which follows from x after the maximal one-turn sequence of moves of A'. Then A' advances by at least h steps with respect to A.*

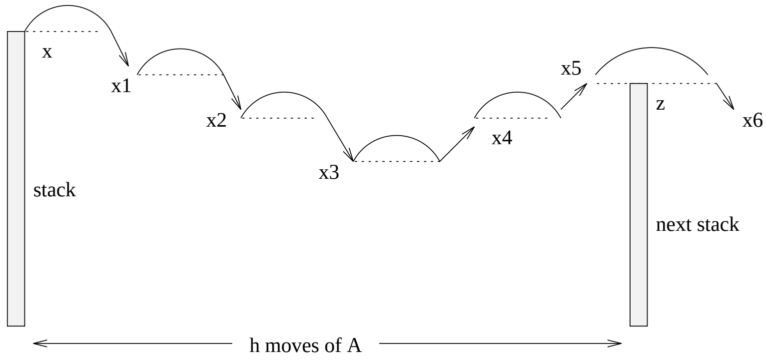

Figure 5: A maximal sequence of type DOWN*·UP*·(DOWN$\vee\epsilon$). It starts at x and ends at $x6$. Each subcomputation denoted by dotted lines is a *shortcut* due to the size of the interval. z is the configuration following x after exactly h moves (assume the simulated automaton A does not stop before). A' advances by at least h steps of A.

Proof. Assume A starts at a configuration x and after exactly h steps arrives at the configuration z. Then each included subcomputation is a shortcut (or a part of a shortcut) due to the size of considered time-interval (see Figure 5).

The accelerated automaton A' will reach the bottom-most position in the stack and will go up using the shortcuts. It is possible that A' will miss z (z would be inside a shortcut) but in any case A' advances at least h steps with respect to A. ∎

Theorem 3.3 *There exists a parallel algorithm for the recognition of deterministic context free languages which takes $O(n^{1-\alpha}\log n)$ time with total work $O(n^{1+\alpha})$.*

Proof. Recall that $h = n^\alpha$. One stage of the algorithm is a simulation of a "long" one-turn sequence of moves of A' in logarithmic time. Lemma 3.2 guarantees that in one stage the time of the simulated DPDA A advances by at least h, except perhaps in the last stage. By a *total configuration* we mean the (surface) configuration together with the contents of the stack. For $k = 1 \ldots \frac{n}{h}$, in the k^{th} stage we compute the total configuration $\langle x, \sigma_k \rangle$, where x is a configuration and σ_k is the next contents of the stack. After k such stages we advance by T_k steps with respect to A, where $T_k \geq \min\{kh, n\}$.

In the k^{th} stage we restrict our computations to the *working area*, that is, the maximal sequence of moves of A (starting form a given total configuration) which together increase the input position and decrease the stack height by at most h (see Figure 6).

In one *stage* of the algorithm a maximal part of a one-turn sequence is simulated which is in the actual working area (see Figure 6).

There are two cases for computations in the $(k+1)^{\text{st}}$ working area. A possible history of a computation in the first case is shown in Figure 6. In this (first) case we follow a sequence of DOWN moves of A' and then a sequence of UP moves and we go outside the working area.

In the second case we have the sequence consisting only of DOWN moves of A'. The height of stack is reduced by h or the stack becomes empty afterwards.

In both cases the time (the number of original steps of A simulated by A' in a single one-turn stage) increases by at least n^α, or terminates in a non-extendible situation.

One stage consists of two substages. The first substage is the computation of the maximal sequence of DOWN moves inside the actual working area. The second substage is the computation of the maximal sequence of push moves of A in the $(k+1)^{\text{st}}$ working area.

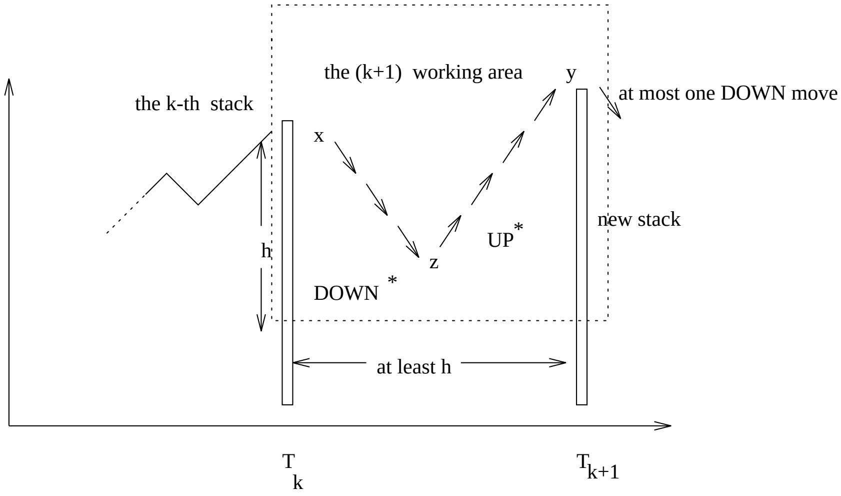

Figure 6: The computations inside the $(k+1)^{\text{st}}$ working area – case 1.

We show only how to implement the first substage and construct the maximal sequence of DOWN configurations in the working area. The construction of the sequence of UP configurations is quite similar.

There are $O(n^\alpha)$ configurations y in the $(k+1)^{\text{st}}$ working area. Let σ_k be the contents of the stack after the k^{th} stage. We consider now only DOWN configurations. Let top_s be the s^{th} symbol of the stack σ_k counting from the top of the stack. If $x' = \langle s, i \rangle$ is a partial configuration and z is a stack element, then we identify the pair $\langle x', z \rangle$ with the configuration $x = \langle s, i, z \rangle$.

If y is a DOWN configuration and $s \leq n^\alpha$, then denote by $next(y, s)$ the pair $\langle \langle \text{DOWN}(y), top_{s-1} \rangle, s-1 \rangle$. The configuration $\langle \text{DOWN}(y), top_{s-1} \rangle$ is realized from y and the next top symbol (after a pop move).

It is easy to see that the maximal sequence of DOWN configurations in the working area is of the form:

$$x_0, next(x_0), next^2(x_0), next^3(x_0), \ldots$$

However, such a sequence can be easily computed in logarithmic time using a squaring technique. The crucial point is that there are $O(n^{2\alpha})$ objects $\langle x, k \rangle$, due to the definition of the $(k+1)^{\text{st}}$ working area and the restriction on the change of the height of the stack. Hence $n^{2\alpha}$ processors are sufficient to compute the maximal sequence of DOWN configurations. The sequence of UP:1 configurations and the additional part of the stack can be computed similarly. The total work results as a product of $n^{2\alpha}$ and the number of stages, which is $O(n^{1-\alpha})$. This completes the proof. ∎

4 Parallel Recognition of Linear Context-Free Languages

A context-free grammar is said to be *linear* if each production has at most one non-terminal on the right side. Any linear context-free language is generated by a grammar where every production is of the form $A \to aB$, $A \to Ba$, or $A \to a$. It was shown in [10] that the problem of recognition of linear context-free languages can be reduced to the *sum-of-path-weights* problem over a *grid graph*, a special kind of directed acyclic graph. The nodes of a grid graph form a square array, and all edges point one position down or to the right. Each edge has a weight, which is a binary relation over the set of nonterminals. The set of such relations forms a *semiring* with the operation being composition of relations. We refer the reader to [10] for more details of how the recognition problem for linear context-free languages reduces to a more general problem related to paths on a grid graph.

In the general sum-of-path-weights problem, each edge in the grid graph has a weight, which is a member of a semiring. The weight of any path is defined to be the product of the weights of the edges that constitute that path, and the problem is to find the sum of the weights of all paths from the *source* (left upper corner of the grid) to the *sink* (bottom right corner). We can assume that every operation in the semiring takes one step, since the semiring has constant size in this application.

The sum-of-path-weights problem can be solved in $O(n^2)$ time sequentially, by dynamic programming, by visiting nodes in topological order, and computing $f(x)$, the sum of the weights of all paths from the source to each node x. The recurrence is:

$$\begin{aligned} f(source) &= \text{identity of the semiring} \\ f(x) &= \sum_{y \to x} f(y) \otimes weight(y, x) \text{ if } x \neq source \end{aligned}$$

We briefly review the parallel algorithm of [2] and [10]. Consider any small square within the grid graph, *i.e.*, the subgraph of all nodes in the square of size d whose upper left corner is (a, b), together with all edges between those nodes, for given a, b, and d. We refer to a node along the top or left edge of the square as an "in-node" and a node along the bottom or right edge as an "out-node" (see Figure 7).

Let $X = X_{a,b,d}$ be the matrix, which we call the *transition matrix* of the subsquare, which relates the values of f on the in-nodes to the values of f on the out-nodes, *i.e.*, if u is an in-node and v is an out-node, $X[u, v]$ is the sum of the weights of all paths from u to v. Then, for each out-node v,

$$f(v) = \sum X[u, v] \otimes f(u)$$

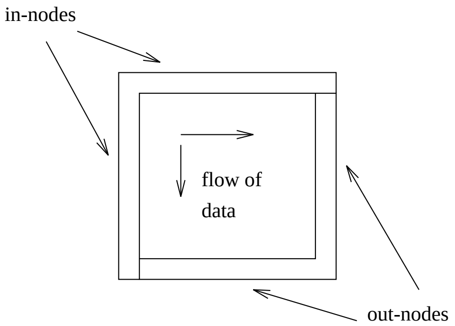

Figure 7: A subsquare and its in-nodes and out-nodes.

where the summation is taken over all in-nodes u.

The following lemma was essentially shown in [2] and [10]. The algorithm behind this lemma is the main procedure in our sublinear time algorithm.

Lemma 4.1 *Assume the matrix X is of shape $d \times d$. Then, using matrix multiplication, X can be computed in $O(\log^2 d)$ time using $d^3/\log^2 d$ processors.*

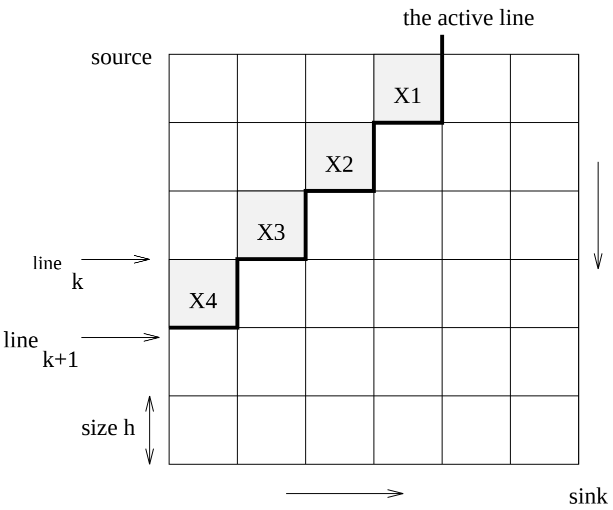

Figure 8: The partition of the matrix into basic subsquares. The values of f on $line_{k+1}$ are computed from the values of f on $line_k$ by using the matrices for all basic subsquares at level k.

Theorem 4.2 *There exists a parallel algorithm for recognition of linear context free languages which requires $O(n^{1-\alpha}\log^2(n))$ time and $O(n^{2+\alpha})$ work.*

Proof. We describe an algorithm which combines the matrix multiplication techniques of [2] and [10] with dynamic programming. The grid graph is partitioned into squares of shape $d \times d$, where $d = n^\alpha$, as shown in Figure 8. The interiors of these squares, which we call *basic subsquares*, are disjoint. Note that there are $O(n^{2-2\alpha})$ basic subsquares. Each basic subsquare is assigned a level, from 0 to $n^{1-\alpha}$, based on the number of steps from the source. If (i, j) is the upper left node of a basic subsquare, its level is $(i + j)/n^{\alpha}$. We define $line_k$ to be the "staircase" shaped set of nodes consisting of all in-nodes of basic subsquares of level k (see Figure 8).

The structure of the proof can be written informally as follows.

Step I: Compute f for all nodes along the top and left edges of the grid graph, *i.e.*, all (i, j) for which $i = 0$ or $j = 0$.

Step II: In parallel, compute the transition matrices for all basic subsquares of the partition.

Step III: Sequentially, for each k from 1 to $n^{1-\alpha}$, compute the values of f on $line_k$ from the values of f on $line_{k-1}$ using transition matrices.

Analysis:

Step I takes $O(\log n)$ time using $n/\log n$ processors.
Step II takes $O(n \log^2 n)$ time using $n^{3\alpha}/\log^2 n$ processors for each basic subsquare by to Lemma 4.1, and therefore $O(\log^2 n)$ time and $n^{2+\alpha}/\log^2 n$ processors altogether.

We note that if the matrix $X_{a,b,d}$ is available, the values of f for all out-nodes can be computed from the values of f for all in-nodes in $O(\log d)$ time using $d^2/\log d$ processors. The computation is reduced to a multplication of a vector by a matrix. Hence Step III needs $O(n^{2+\alpha})$ work altogether. ∎

References

[1] A. Aho, J. Hopcroft, J. Ullman, *The Design and analysis of computer algorithms*, Addison-Wesley (1974).

[2] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. L. Miller, and S-H. Teng. Constructing trees in parallel, *Proc. 1st Symp. on Parallel Algorithms and Architectures* (1989), pp. 499–533.

[3] L. Banachowski, A. Kreczmar, W. Rytter, *Analysis of algorithms and data structures*, Addison-Wesley (1991).

[4] M. Chytil, M. Crochemore, B. Monien, and W. Rytter, On the parallel recognition of unambiguous context-free languages, *Theoretical Computer Science* **81**, pp. 311–316.

[5] A. Gibbons, W. Rytter, *Efficient parallel algorithms*, Cambridge University Press (1988).

[6] M. A. Harrison, *Introduction to formal language theory*, Addison Wesley (1978).

[7] J. Hopcroft, J. Ullman, Automata, languages and computations Addison-Wesley (1979).

[8] L. L. Larmore, W. Rytter, An Optimal sublinear time parallel algorithm for some dynamic programming problems, *Information Processing Letters* **52** (1994), pp. 31–34.

[9] B. Monien, W. Rytter, and H. Schapers, Fast recognition of deterministic CFL's with a smaller number of processors, *Theoretical Computer Science* **116** (1993), pp. 421–429.

[10] W. Rytter, On the Parallel computation of costs of paths on a grid graph, *Information Processing Letters* **29** (1988), pp. 71–74.