DIMACS Technical Report 97-50

$RUSPACE(\log n) \subseteq DSPACE(\log^2 n/\log\log n)$[1]

by

Eric Allender[2,3]
Department of Computer Science
Rutgers University
Piscataway, NJ 08855
USA
allender@cs.rutgers.edu

Klaus-Jörn Lange[4]
Wilhelm-Schickard Institut für Informatik
Universität Tübingen
Sand 13
D-72076 Tübingen
Germany
lange@informatik.uni-tuebingen.de

[1]A preliminary version of this work appeared in Proc. 7th Annual International Symposium on Algorithms and Computation (ISAAC'96), Lecture Notes in Computer Science 1178, pp. 193–202.
[2]Permanent Member
[3]Supported in part by NSF grant CCR-9509603.
[4]Supported in part by NSF grant CCR-9509603.

DIMACS is a partnership of Rutgers University, Princeton University, AT&T Labs, Bellcore, and Bell Labs.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

We present a deterministic algorithm running in space $O\left(\log^2 n/\log\log n\right)$ solving the connectivity problem on strongly unambiguous graphs. In addition, we present an $O(\log n)$ time-bounded algorithm for this problem running on a parallel pointer machine.

1 Introduction

A central challenge facing complexity theory is to relate determinism and nondeterminism. Our inability to exhibit the precise relationship between these two notions motivates the investigation of intermediate notions such as symmetry and unambiguity. In this paper we concentrate on unambiguity in space-bounded computation, and present improved deterministic and parallel simulations.

Recently, surprising results have indicated that "symmetric" space bounded computation is weaker than nondeterminism. In particular, symmetric logspace has been shown to be contained in parity logspace [14], in SC^2 [20], and in $DSPACE(\log^{4/3} n)$ [2]. None of these upper bounds is known to hold in the nondeterministic case. If we consider these questions for space bounded unambiguous classes, we are confronted with the fact that there are several ways to define notions of unambiguity that apparently do not coincide [4]. In this paper we will concentrate on the notions of unambiguity (in the sense of unique existence of accepting computation paths), and of strong unambiguity (in the sense of uniqueness of computations between any pair of configurations). Our results apply equally well to an intermediate notion of unambiguity, called "reach unambiguity", which will be described in more detail in section 3; we argue there that reach unambiguity is the most useful of these notions. This yields the three classes $USPACE(\log n)$, $RUSPACE(\log n)$, and $StUSPACE(\log n)$.

By definition, $USPACE(\log n)$ is a subclass of parity logspace; this is not known to hold for $NSPACE(\log n)$ (although see [11]); there is no additional nontrivial containment known for $USPACE(\log n)$. However, $RUSPACE(\log n)$ is contained in SC^2, since reach-unambiguous logspace languages can be accepted by deterministic auxiliary pushdown automata in polynomial time [4, 6]. Still, it was unknown whether there are $o(\log^2 n)$ space algorithms for strongly unambiguous logspace languages. We answer this question affirmatively by showing that $RUSPACE(\log n)$ (and hence also $StUSPACE(\log n)$) is contained in $DSPACE(\log^2 n/\log\log n)$.

Since $RUSPACE(\log n)$ is a subclass of $DAuxPDA\text{-}TIME\left(n^{O(1)}\right)$ we know that there are logtime $CROW$-algorithms for the elements of $RUSPACE(\log n)$ [8]. To give better relative upper bounds on the complexity of $RUSPACE(\log n)$ it is interesting to consider intermediate classes between $DSPACE(\log n)$ and $DAuxPDA\text{-}TIME\left(n^{O(1)}\right)$.

Trying to find these classes with sequential models could be difficult, since the usual restrictions of a pushdown store (e.g. the one-turn property, or using a counter instead of a push down) all collapse to logspace. Here, parallel machine models seem helpful, leading to two intermediate classes: the parallel pointer machine [7, 16] and the $OROW$-PRAM [23]. As a consequence of our main result we get $OROW$ algorithms for the elements of $RUSPACE(\log n)$ taking time $O(\log^2 n/\log\log n)$. We are also able to show that all sets in $RUSPACE(\log n)$ are accepted in logarithmic time on a parallel pointer machine. This latter containment is somewhat surprising, because there are characterizations in terms of parallel programs indicating that the class $PPM\text{-}TIME(\log n)$ is rather close to $DSPACE(\log n)$ [18].

2 Preliminaries

We assume the reader to be familiar with the basic notions of complexity theory (e.g. [12]).

We refer the reader to the survey article of Karp and Ramachandran [15] for coverage of the many varieties of parallel random access machines and their relationship to sequential classes. Let us remark here, that we deal in this paper only with algorithms and classes using PRAMs with a polynomial number of processors. The notion of a *CROW*-PRAM was introduced by Dymond and Ruzzo [8]. This model of PRAM offers many advantages over other models such as CRCW or CREW PRAMs. For instance, although the complexity class NC has a simple characterization on all of these PRAM models (namely: polynomial number of processors and polylog time), SC is known to have a simple characterization on the CROW model (logarithmically many processors, quasipolynomial time) but not on the other types of PRAM. More information about this and other results detailing the advantages of the CROW model can be found in [10].

CROW-PRAMs need only logarithmic time to recognize any given language in $DSPACE(\log n)$; There are two important ways to restrict *CROW*-PRAMs and still maintain this property. One way is to restrict the concurrent read access to the global memory, which leads to the *OROW*-PRAMs of Rossmanith [23]. The other way is to restrict the arithmetical capabilities of the instruction set leading to *rCROW*-PRAMs and to parallel pointer machines [7, 16].

3 Unambiguity

A concept intermediate in power between determinism and nondeterminism is *Unambiguity*. A nondeterministic machine is said to be unambiguous, if for every input there exists at most one accepting computation. This leads to the classes UP and $USPACE(\log n)$; we have $P \subseteq UP \subseteq NP$ and $DSPACE(\log n) \subseteq USPACE(\log n) \subseteq NSPACE(\log n)$. The notion of unambiguity should be distinguished from that of *uniqueness*, which uses the unique existence of an accepting path not as a restriction but as a tool. The resulting language classes $1NSPACE(\log n)$ and $1NP$ consist of languages defined by machines that accept their inputs if there is exactly one accepting path. Thus, the existence of two or more accepting computations is not forbidden, but simply leads to rejection. In the polynomial time case we have $Co\text{–}NP \subseteq 1NP$ [3]. In the logspace case inductive counting [13, 25] shows $1NSPACE(\log n) = NSPACE(\log n)$.

A more restrictive form of unambiguity is *Strong Unambiguity*. A nondeterministic machine is said to be strongly unambiguous, if for *every* pair of configurations there exits at most one computational path connecting these configurations. An ordinary unambiguous machine makes this restriction only for the initial and the accepting configurations of the machine.

While these two concepts coincide (yielding the class UP) in the case of time bounded computations, this is not known to be true in the space bounded case. There we end up with an additional class $StUSPACE(\log n)$ which is located between $DSPACE(\log n)$ and $USPACE(\log n)$. In fact, there are several more versions of unambiguity that are not known to coincide (depending for instance on whether or not there can be more than one accepting configuration, see [4]), but there is just one of these other versions that we will consider here, because this version has recently assumed greater importance than some of the others: *Reach-Unambiguity.*

A nondeterministic machine is said to be reach-unambiguous if for every pair of configurations *that are reachable from the start configuration*, there exists at most one computational path connecting these configurations. Reach-unambiguous machines may violate the condition of strong unambiguity for unreachable configurations. This yields the class *RUSPACE*$(\log n)$; note that *StUSPACE*$(\log n) \subseteq$ *RUSPACE*$(\log n) \subseteq$ *USPACE*$(\log n)$. The next few paragraphs present motivation for studying reach-unambiguity.

In the time-bounded setting, problems such as factoring and primality have efficient unambiguous algorithms but are not known to possess deterministic algorithms with a comparable running time [9]. In the space-bounded setting, until recently there had not been any corresponding computational problem whose complexity was best modeled by space-bounded unambiguity. Recently, however, Lange presented a problem that is complete for *RUSPACE*$(\log n)$ [17]; this is the first explicit presentation of a problem in *USPACE*$(\log n)$ that is not known to be in *DSPACE*$(\log n)$. (Completeness is a tool that is not often available in studying unambiguous classes. None of *UP*, *USPACE*$(\log n)$, or *StUSPACE*$(\log n)$ is known or believed to have complete sets.)

No problem is known to be in *StUSPACE*$(\log n)$ that is not known to be in *DSPACE*$(\log n)$. Nonetheless, there is an important *class* of languages with this property: The class of unambiguous linear languages, *ULIN*, is contained in *StUSPACE*$(\log n)$ [4], and it is not known to be contained in *DSPACE*$(\log n)$. To date, however, no explicit example of a language in *ULIN* has been exhibited for which a *DSPACE*$(\log n)$ algorithm is not known. (Note in this regard that the linear context free languages are *NSPACE*$(\log n)$–complete.)

The algorithm presented in this paper was originally devised with the class *StUSPACE*$(\log n)$ in mind, but it applies equally well for all classes for which the tree-unfoldings of the subgraphs of reachable configurations have polynomial size. This is the case for *RUSPACE*$(\log n)$ but not for *USPACE*$(\log n)$. Thus *RUSPACE*$(\log n)$ has the property that (1) all currently-known completeness results and also (2) all nontrivial upper bounds that are known for unambiguous logspace classes hold for *RUSPACE*$(\log n)$.

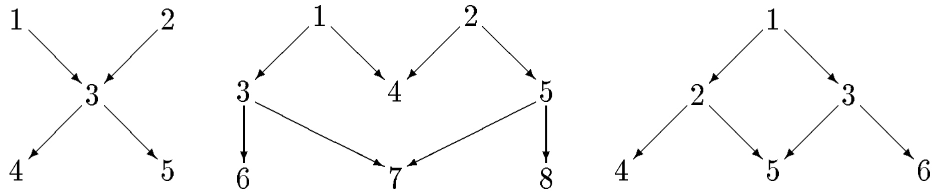
Logspace classes are closely connected to graph accessibility problems. These connectivity problems in directed graphs, undirected graphs, and in trees are complete for *NSPACE*$(\log n)$, *SSPACE*$(\log n)$, and *DSPACE*$(\log n)$, respectively. For unambiguous classes the corresponding connectivity problems do not seem to be complete (except for the important case of *RUSPACE*$(\log n)$, as noted previously). Nevertheless it is useful to explain these notions in terms of directed graphs, since this will make the demonstration of our algorithms easier.

Let $G = (V, E)$ be a directed acyclic graph. If G has n nodes we assume V to be $\{1, 2, \cdots, n\}$ and we will be interested in the existence of a path leading from 1 to n. For each pair of nodes (x, y) let $d(x, y)$ be the length of the shortest path between x and y. If x and y are not connected, $d(x, y)$ is infinite; $d(x, x)$ is 0. In the following we will work with complete (but not necessarily balanced) binary graphs; that is, each node of G is either a leaf with no outgoing edges, or an inner node with two outgoing edges. We assume the leaves to be accepting or rejecting. This is determined by a mapping $\phi : V \longrightarrow \{+, -, i\}$, which takes the value $+$ for accepting leaves, $-$ for rejecting leaves, and i for interior nodes. In particular, 1 is an inner node (unless the graph has only one vertex) and we assume n to be the only accepting leaf, since we are only interested in paths from 1 to n. The two successors of an inner node x will be denoted by $L(x)$ and $R(x)$.

Let $N(x, y)$ be the number of paths from x to y in G. For $x \in V$ let $T(x) := \{y \in V | N(x, y) > 0\} \cup \{x\}$ be the set of all nodes reachable from x.

G is called *unambiguous* if there is at most one path from 1 to n, i.e.: if $N(1, n) \leq 1$. G is called *strongly unambiguous* or a *Mangrove* if for any pair (x, y) of nodes there is at most one path leading from x to y, i.e.: if $\forall_{x,y \in V}\ N(x, y) \leq 1$. Although a mangrove does not need to be a tree, for each x the subgraph of G induced by $T(x)$ is indeed a tree and the same is true for the set of all nodes from which x can be reached. G is called *reach-unambiguous* if $N(1, i) \leq 1$ for each i.

The following three examples show different versions of unambiguous graphs. The first one is a very simple mangrove with a positive result since there is a path from node 1 to node 5. The second one is also a mangrove, but a negative one, since there is no path from 1 to 8. The third graph is unambiguous and positive, since there is exactly one path from 1 to 6, but it is not reach-unambiguous, since there are two different paths from node 1 to node 5.



Typical examples of mangroves are butterfly graphs, and the following family of graphs introduced by Klaus Reinhardt [21]: $R_n := (V_n^r, E_n^r)$ with $V_n^r := \{0, 1, 2, \cdots, n-1\} \times \{1, 2, \cdots, n\}$ and $E_n^r = \{(\langle i, j\rangle, \langle i, j+1\rangle) | 0 \leq i \leq n-1, 1 \leq j < n\} \cup \{(\langle i, j\rangle, \langle i + j \bmod n, n\rangle) | 0 \leq i \leq n-1, 1 \leq j < n\}$. (Observe that these graphs are not complete binary graphs.)

By definition a Turing machine is unambiguous if for each input word the reachability graph of its configurations is unambiguous. It is strongly unambiguous if for every input this graph forms a mangrove. It is reach-unambiguous if for every input this graph is reach-unambiguous.

4 Contracting Mangroves

The class *RUSPACE*$(\log n)$ has been shown to be contained in *DAuxPDATIME*$(n^{O(1)})$ by traversing the tree of reachable configurations [4]. This tree can be searched with the help of a push-down store in polynomial time. It can thus be seen that $RUSPACE(\log n) \subseteq SC^2$, since every problem in the class $DAuxPDA\text{-}TIME\left(n^{O(1)}\right)$ is reducible to a deterministic context free language [24], and all such languages are in SC^2 via the algorithm of Cook [6]. Elements of Cook's algorithm were later used by Dymond and Ruzzo [8] and independently by Monien et al [19] to show $DCFL \subseteq CROW\text{-}TIME(\log n)$ (see also [10]). As a consequence, the elements of $RUSPACE(\log n)$ possess logtime algorithms running on a CROW–PRAM. But these algorithms for mangroves, generated by composing the constructions in [4] with those of [8, 10, 19], are rather complicated. Below, we give a very simple logtime algorithm that runs on a parallel pointer machine (which is a restricted CROW-PRAM). Our main result (Theorem 2) is proved using a slightly more sophisticated variant of this same approach.

Theorem 1

$$RUSPACE(\log n) \subseteq PPM\text{-}TIME(\log n)$$

Proof: We first describe the parallel algorithm and then indicate how to run it on a parallel pointer machine. Assume first that A is a strongly unambiguous logspace machine and let $G = (V, \phi, L, R)$ be its configuration mangrove induced by some input w of size n. For each $x \in V$ the tree $T(x)$ is of polynomial size in n. A first approach would be to attempt to perform pointer jumping on $T(x)$. Unfortunately, the pointers are directed toward the leaves instead of toward the root. The alternative is to perform the shunt operation (see e.g. [15]). This would need backpointers directed towards the root, which are usually provided by establishing an Euler tour through a tree. A mangrove is sufficiently different from a tree, however, that this technique also does not work. The simple way out of this is to perform the shunt operation not by the parent node of a leaf but by its grandparent node, in parallel in each tree. Of course, these operations destroy any tree structure unless they are synchronized. But obviously the property of being a mangrove is invariant under the parallel application of shunt operations. Hence there is no need to do some sophisticated arrangement of the working processors as in [1, 5]. (It should be remarked that the shunt operation preserves the outdegrees of the nodes to be either zero or two. This is not true for indegrees.) This is explained in more detail below.

For each $x \in V$ that is not a leaf, i.e. $\phi(x) = i$, we first check whether one of its children is an accepting leaf, in this case we set $\phi(x) := +$. (Here, note that the mapping ϕ is not a constant, but is an array that is stored in global memory and can be updated.) If both children are rejecting we set $\phi(x) := -$. Otherwise, one of the children has to be an inner node. If $\phi(L(x)) = i$ we check if the left grandson via the left subtree is a rejecting leaf. In that case we set $L(x) := R(L(x))$. If this was not the case we see whether the right grandson is a rejecting leaf which would lead to the assignment $L(x) := L(L(x))$. This is then repeated for the right son of x.

It is important to note that for each node x and reachability tree $T(x)$, one application of this operation in parallel results in decreasing the size of the tree by a constant fraction, since (1) each grandparent of a leaf of $T(x)$ loses one child and a grandchild, and (2) at least one-seventh of the nodes in a complete binary tree are grandparents of leaves. After performing this parallel transformation $O(\log n)$ times, there is no inner node left; for each $x \in V$ we have either $\phi(x) = +$ or $\phi(x) = -$. The existence of an accepting path is then equivalent to $\phi(1) = +$.

Formally, the algorithm is expressed by the following statements:

Algorithm 1

```
for j = 1, 2, ···, O(log n) do
    for all x ∈ V do in parallel
        if φ(x) = i then
            if φ(L(x)) = + or  φ(R(x)) = + then φ(x) := +;
            if φ(L(x)) = − and  φ(R(x)) = − then φ(x) := −;
            if φ(L(x)) = i then
                if φ(L(L(x))) = − then  L(x) := R(L(x))
                else
                    if φ(R(L(x))) = − then  L(x) := L(L(x));
            if φ(R(x)) = i then
                if φ(L(R(x))) = − then  R(x) := R(R(x))
                else
                    if φ(R(R(x))) = − then  R(x) := L(R(x));
if φ(1) = + then accept
else reject
```

We now briefly indicate how to implement this on a parallel pointer machine. Obviously, the algorithm itself is already suitable for a PPM. It remains only to show how to build the mangrove of configurations in logarithmic time, given the input word. As in the corresponding construction of Cook and Dymond [7] the initial PPM unit starts to build in logarithmic time a tree of logarithmic depth such that each leaf unit corresponds to a configuration of A on w. Then the leaves are interconnected according to the successor relation of A. But instead of one pointer leading to a successor, each leaf unit will now have two pointers L and R representing the two subtrees hanging below an inner node of a mangrove.

It remains only to observe that the algorithm in its current form also works if we assume only that A is reach-unambiguous, instead of strongly unambiguous. To see this, note that our analysis makes crucial use of the fact that, for each x in a mangrove, $T(x)$ is a tree. If we assume only that we are dealing with a reach-unambiguous graph, then although $T(x)$ may be an arbitrary directed acyclic graph for *unreachable* nodes x, it is important to note that $T(x)$ will still be a tree for every node x reachable from the start configuration. Thus the same analysis carries over. □

Let us remark here that the number of PPM units, i.e. processors, is linear in the size of the mangrove. For the special case of *ULIN*, this leads to a quadratic number of processors. The best known sequential algorithm for solving membership questions in *ULIN* uses quadratic time.

Our algorithm makes intensive use of concurrent reads and therefore does not pertain to owner read PRAMs. It works via $O(\log n)$ applications of the operations of searching and pruning trees of depth 2. Increasing this depth to $O(\log n)$ is the first step in reducing the number of iterations to $O(\log n/\log\log n)$. This yields an algorithm using $o(\log^2 n)$ space (or parallel time).

Theorem 2

$$RUSPACE(\log n) \subseteq DSPACE(\log^2 n/\log\log n)$$

PROOF: Again, assume first that A is a strongly unambiguous logspace machine and let w be an input of length n. This yields a reachability problem in a mangrove $G = (V, \phi, L, R)$ of polynomial size.

We will now construct a mangrove $G' = (V, \phi', L', R')$ of smaller size. Let $t \geq 1$ be an integer that determines the rate of contraction of G' compared to G. We will fix the value of t later. First we map each node x to one of its successors which we call $f(x)$.

If in the following procedures an accepting leaf (i.e. a node y with $\phi(y) = +$) is found, the search is stopped, and we set $\phi'(x) := +$ and $f(x) := x$. For each $x \in V$ we search the tree $T_t(x) := \{y \in V | d(x, y) \leq t\}$ of all nodes of distance to x not greater than t.

Let z be a pointer initialized to x. Consider the set M of all nodes y in $T_t(z)$ with $d(z, y) = t$ and $\phi(y) = i$, i.e. of all leaves of $T_t(z)$ that are not leaves in $T(z)$. If M is empty, that is if $T(z) = T_t(z)$ then set $\phi'(x) := -$ and $f(x) := x$. If M is nonempty we replace z by the least common ancestor z' of M in $T_t(z)$. If $z = z'$ we stop the procedure for x and set $f(x) := z$ and $\phi'(x) := i$. Otherwise we replace z by z' and continue the process.

This algorithm computing f and ϕ' in a more formal way looks as follows, where $LCA(M)$ denotes the least common ancestor of a set M of nodes in a tree:

Algorithm 2

> $z := x$;
> $M := \{y \in T_t(z)\,|d(z, y) = t, \phi(y) = i\}$;
> **while** $+ \notin \phi(T_t(z))$ **and** $M \neq \emptyset$ **and** $LCA(M) \neq z$ **do**
> **begin**
> $z := LCA(M)$;
> $M := \{y \in T_t(z)\,|d(z, y) = t, \phi(y) = i\}$
> **end**
> **if** $+ \in \phi(T_t(z))$
>   **then** $\phi'(x) := +; f(x) := x$
>   **else if** $M = \emptyset$
>   **then** $\phi'(x) := -; f(x) := x$
>   **else** $\phi'(x) := i; f(x) := z$

After this process, for each $x \in V$ either $\phi'(x) = +$, i.e. an accepting leaf has been found in $T(x)$, or $\phi'(x) = -$, i.e. $T(x)$ has been totally searched and contains only rejecting leaves, or $\phi'(x) = i$. In this case, the node $f(x)$ has the property that both subtrees of $f(x)$ contain nodes having distance greater than t from $f(x)$. That is, both subtrees of $f(x)$ are of height not smaller than t, and, since both are complete binary trees, both of them have size not less than $2t + 1$.

Clearly, the computation of f can be done in space $t + O(\log n)$. (Note that M need not be stored explicitly.) We now construct G' by setting $L'(x) := f(L(f(x)))$ and $R'(x) := f(R(f(x)))$ for each x with $\phi'(x) = i$. We have for each $x \in V$ with $\phi'(x) = i$:

$$(P1)\ \ |T(L(f(x)))| \geq 2t + 1 \text{ and } |T(R(f(x)))| \geq 2t + 1.$$

Furthermore, the construction implies

$$(P2)\ \ f(f(x)) = f(x) \text{ and } (P3)\ \ \phi'(x) = \phi'(f(x))$$

for every $x \in V$.

Now let $T'(x)$ be the tree below x in G'. Although there may be nodes in V that are not finished (i.e. $\phi'(x) = i$) and that are not contracted, (i.e. $f(x) = x$) the mangrove in total is smaller by a factor of t; we claim for each $x \in V$ with $\phi'(x) = i$:

$$|T'(x)| \leq |T(x)| / t$$

> Proof of the claim: Let $L'(z)$ be a left leaf in the tree $T'(x)$. That is, $\phi'(z) = i$ and $\phi'(L'(z)) \neq i$. Thus
>
> $$\begin{aligned} \phi'(L(f(z))) &= \phi'(f(L(f(z)))) && \text{by (P3)} \\ &= \phi'(L'(z)) && \text{by definition of } L' \\ &\neq i && \text{by assumption} \end{aligned}$$
>
> That is $L(f(z))$ is a leaf in T'. Inspection of Algorithm 2 shows that, for any node v such that $\phi'(v) \neq i$, we have $f(v) = v$. Hence $L(f(z)) = f(L(f(z))) = L'(z)$. Thus by (P1),
>
> $$|T(L'(z))| = |T(L(f(z)))| \geq 2t + 1.$$
>
> Hence for each left leaf $L'(z)$ in $T'(x)$ there are at least $2t$ nodes that are removed from $T(x)$. The same holds for right leaves. Hence $T(x)$ is not smaller than $2t$ times the number of leaves of $T'(x)$. Since at least half of the elements of a complete binary tree are leaves, the result follows.

Repeating this process of shrinking G for $O(\log_t n)$ phases results in setting $\phi'(1)$ to $+$ or $-$, thus yielding the solution. Setting $t := \log n$ we obtain $O(\log n / \log\log n)$ phases, each of which uses $O(\log n)$ space, to obtain a procedure to search a mangrove using $O(\log^2 n / \log\log n)$ space.

Our algorithm will not run efficiently if it is started on any node x such that $T(x)$ is not a tree (or, more generally: if the tree that results from "unfolding" the graph $T(x)$ is not of polynomial size). However, if our graph of configurations is reach-unambiguous, then we only run our algorithm for nodes x for which $T(x)$ is a tree. Hence, the same analysis suffices also for any problem in $RUSPACE(\log n)$.

Somewhat more generally, if the unfolding of $T(x)$ has size bounded by $2^{r(n)}$ (where r may be superlogarithmic), then our algorithm runs in space $O(r(n)\log n/\log\log n)$. $\Box$

We remark here, that the resulting algorithm is in general not polynomial time-bounded.

Corollary 3

$$ULIN \subseteq DSPACE(\log^2 n/\log\log n)$$

All of $DSPACE(f)$ can be recognized by $OROW$-PRAMs in $O(f)$ steps. But generally these algorithms need $c^{O(f)}$ many processors, which in our case would be superpolynomial. But due to the recursive or iterative structure of our algorithm a polynomial number suffices, as we now show.

Corollary 4

$$RUSPACE(\log n) \subseteq OROW\text{-}TIME\left(\log^2 n/\log\log n\right)$$

Proof: The PRAM works in $\log n/\log\log n$ phases, each taking $O(\log n)$ steps. In each phase the work of a logspace Turing machine is simulated in logarithmic time by an $OROW$-PRAM with a polynomial number of processors [23]. There is a slight difference between what we need here and what is provided by the simulation in [23], in that here we don't simulate merely an accepting machine, but a machine producing some output. This output is stored in global memory. In order to use this output as input to the next phase, we first have to distribute it to polynomially many processors in a tree-like way in $O(\log n)$ steps. In total this costs $O(\log^2 n/\log\log n)$ steps on an $OROW$-PRAM. The number of processors corresponds to the number of configurations of the simulated logspace machine and hence is polynomial. $\Box$

We mention in passing that this result, applied to the unambiguous linear languages, leads to algorithms using $O(n^2)$ processors since this is the size of the corresponding mangrove.

We end this section by remarking on a generalization of our main theorem. It is well-known that $NTISP(f,g)$ is a subset of $DSPACE(g\log f)$. (Here, $NTISP(f,g)$ is the set of all languages accepted by nondeterministic $O(g)$ space-bounded Turing machines in time $O(f)$.) Define the corresponding class $RUTISP(f,g)$ to be the class of all languages accepted by reach-unambiguous Turing machines that are simultaneously $O(g)$ space- and $O(f)$ time-bounded. Our space-efficient simulation then yields the inclusion: $RUTISP(f,g) \subseteq DSPACE(g \cdot \log f/\log g)$.

5 Discussion and open questions

The most basic open question is, of course, to clarify the relationships among $RUSPACE(\log n)$, $StUSPACE(\log n)$, and $DSPACE(\log n)$. It might very well be that these three classes coincide. Indeed, there seem to be no drastic consequences implied by such a collapse. A first step in this direction would be to exhibit a logtime *OROW*-algorithm for *ULIN* (or even for *StUSPACE*$(\log n)$).

Another open issue concerns the class *USPACE*$(\log n)$. Is it possible to show that it is contained in SC^2 or to give an $o(\log^2 n)$ space algorithm, as has been possible for symmetric logspace and the class *RUSPACE*$(\log n)$? In this regard, it is interesting to call the reader's attention to a very recent result in [22]: *USPACE*$(\log n)$/poly = *NSPACE*$(\log n)$/poly. Thus any improved upper bound on the complexity of *USPACE*$(\log n)$ will have strong implications on the complexity of *NSPACE*$(\log n)$.

A third line of investigation is to consider these questions for polynomial time bounded auxiliary pushdown automata where results analogous to our Theorems 1 and 2 are probably harder to obtain. The complexity of *UCFL*, the class of unambiguous context free languages, is uncertain since we do not know whether it is complete for the strongly unambiguous auxiliary pushdown class, just as we don't know whether *ULIN* is complete for *StUSPACE*$(\log n)$. Since $RUSPACE(\log n) \subseteq DAuxPDA\text{-}TIME\left(n^{O(1)}\right)$ and since *DCFL* is complete for the later class, we know $RUSPACE(\log n) \subseteq LOG(UCFL)$. It is not known whether *UCFL* is also hard for *USPACE*$(\log n)$. On the other hand it might well be the case that *UCFL* is contained in SC^2.

Acknowledgments

We thank DIMACS for the use of office space where much of this work was done while the second author visited Rutgers University. Portions of this work were also done while the first author was on sabbatical leave at the Institute of Mathematical Sciences, Chennai, India.

References

[1] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. In *VLSI Algorithms and Architectures, Proc. 3rd Aegean Workshop on Computing*, number 319 in LNCS, pages 81–90. Springer, 1988.

[2] R. Armoni, A. Ta-Shma, A. Wigderson and S. Zhou. $SL \subseteq L^{\frac{4}{3}}$. In *Proc. 29th Annual ACM Symp. on Theory of Computing*, 1997, pp. 230–239.

[3] A. Blass and Y. Gurevich. On the unique satisfiability problem. *Inform. and Control*, 55:80–88, 1982.

[4] G. Buntrock, B. Jenner, K.-J. Lange, and P. Rossmanith. Unambiguity and fewness for logarithmic space. In *Proc. of the 8th Conference on Fundamentals of Computation Theory*, number 529 in LNCS, pages 168–179, 1991.

[5] R. Cole and U. Vishkin. Approximate parallel scheduling, part i: the basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comp.*, 17:128–142, 1988.

[6] S. Cook. Deterministic CFL's are accepted simultaneously in polynomial time and log squared space. In *Proc. of the 11th Annual ACM Symp. on Theory of Computing*, pages 338–345, 1979.

[7] S. Cook and P. Dymond. Parallel pointer machines. *Computational Complexity*, 3:19–30, 1993.

[8] P. Dymond and W. Ruzzo. Parallel RAMs with owned global memory and deterministic context-free language recoginition. In *Proc. of 13th International Colloquium on Automata, Languages and Programming*, number 226 in LNCS, pages 95–104. Springer, 1986.

[9] M. Fellows and N. Koblitz. Self-witnessing polynomial-time complexity and prime factorization. In *Proc. of the 7th IEEE Structure in Complexity Conference*, pages 107–110, 1992.

[10] H. Fernau, K.-J. Lange, and K. Reinhardt. Advocating ownership. In *Proc. of 17th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 1180 in LNCS, pages 286–297. Springer, 1996.

[11] A. Gál and A. Wigderson. Boolean vs. arithmetic complexity classes: randomized reductions. *Random Structures and Algorithms*, 9:99–111, 1996.

[12] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Language, and Computation.* Addison-Wesley, Reading Mass., 1979.

[13] N. Immerman. Nondeterministic space is closed under complementation. *SIAM J. Comp.*, 17:935–938, 1988.

[14] M. Karchmer and A. Wigderson. On span programs. In *Proc. of the 8th IEEE Structure in Complexity Theory Conference*, pages 102–111, 1993.

[15] R.M. Karp and V. Ramachandran. A Survey of Parallel Algorithms for Shared-Memory Machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A*, pages 869–941. Elsevier, Amsterdam, 1990.

[16] T. Lam and W. Ruzzo. The power of parallel pointer manipulation. In *Proc. of the 1st ACM Symposium on Parallel Algorithms and Architectures (SPAA'89)*, pages 92–102, 1989.

[17] K.-J. Lange. An unambiguous class possessing a complete set. In *Proc. of the 14th STACS*, number 1200 in LNCS, pages 339–350. Springer, 1997.

[18] K.-J. Lange and R. Niedermeier. Data-independences of parallel random access machines. In *Proc. of 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 761 in LNCS, pages 104–113. Springer, 1993.

[19] B. Monien, W. Rytter, and H. Schäpers. Fast recognition of deterministic cfl's with a smaller number of processors. *Theoret. Comput. Sci.*, 116:421–429, 1993. Corrigendum, 123:427,1993.

[20] N. Nisan. $RL \subseteq SC$. *Computational Complexity*, 4:1–11, 1994.

[21] K. Reinhardt. Personal communication.

[22] K. Reinhardt and E. Allender. Making nondeterminism unambiguous. To appear in *Proc. 38th Annual IEEE Symposium on Foundations of Computer Science*, 1997.

[23] P. Rossmanith. The owner concept for PRAMs. In *Proc. of the 8th STACS*, number 480 in LNCS, pages 172–183. Springer, 1991.

[24] I. Sudborough. On the tape complexity of deterministic context-free languages. *J. Assoc. Comp. Mach.*, 25:405–414, 1978.

[25] R. Szelepcsényi. The method of forcing for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.