An $O(\log n \log\log n)$ Space Algorithm for Undirected st-Connectivity

[Extended Abstract]

Vladimir Trifonov*
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712-0233, USA
vladot@cs.utexas.edu

ABSTRACT

We present a deterministic $O(\log n \log\log n)$ space algorithm for undirected st-connectivity. It is based on the deterministic EREW algorithm of Chong and Lam [6] and uses the universal exploration sequences for trees constructed by Koucký [13]. Our result improves the $O(\log^{4/3} n)$ bound of Armoni et al. [2] and is a big step towards the optimal $O(\log n)$. Independently of our result and using a different set of techniques, the optimal bound was achieved by Reingold [18].

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*

General Terms

Theory, Algorithms

Keywords

undirected st-connectivity, space complexity, space-bounded computation, space-efficient simulation of parallel algorithms

1. INTRODUCTION

The problem we are concerned with is st-connectivity in an undirected graph G with n vertices (USTCONN), i.e. given two vertices s and t of G we want to answer the question whether there is a path between s and t. This is one of the most basic graph problems with applications ranging from image processing and VLSI design to solving more complex graph problems. Furthermore, st-connectivity plays an important role in complexity theory because directed st-connectivity (STCONN) is **NL**-complete [20] and USTCONN is **SL**-complete [14].

Linear time and space sequential algorithm for solving even the harder STCONN problem have been known for a long time [22]. The problem of developing more efficient space and parallel algorithms was posed.

If we allow one-sided randomness, the result of Aleliunas et al. [1] shows that USTCONN can be solved in $O(\log n)$ space. The starting point of deterministic space-efficient sequential algorithms is the $O(\log^2 n)$ space algorithm for STCONN of Savitch [20]. For a long time this was the best result even for USTCONN. The space bound for undirected graphs was first improved by Nisan et al. [16] to $O(\log^{3/2} n)$ and then to $O(\log^{4/3} n)$ by Armoni et al. [2]. Both of these results depend on the efficient construction of universal traversal sequences by Nisan [15]. Finally, simultaneous to our result and using different set of techniques, the space complexity of USTCONN was shown by Reingold [18] to be $O(\log n)$.

Developing efficient parallel algorithms for USTCONN has a very rich history. The situation here is complicated further by the existence of multiple models of parallel computation. The models from the PRAM family are generally considered to be of great theoretical value. The results of [10, 19, 4, 11, 12] are concerned with the CREW PRAM model, [21, 3, 7, 8] with the CRCW PRAM model, and [9, 6, 5, 17] with the EREW PRAM. The state of the art in parallel algorithms for USTCONN are the results of Chong, Han, and Lam [5], which shows that the problem can be solved on the EREW PRAM in $O(\log n)$ time with $O(m+n)$ processors, and Pettie and Ramachandran [17], which demonstrates a randomized EREW PRAM algorithm running in time $O(\log n)$ with optimal number of processors. The algorithms of [5] and [17] actually solve the harder problem of finding the minimum spanning tree of a weighted graph.

The starting point of our algorithm is the $O(\log n \log\log n)$ time deterministic EREW PRAM algorithm with $O(m+n)$ processors of Chong and Lam [6], which we call the CL algorithm. This parallel algorithm can be trivially simulated sequentially in linear space. We use the sequential algorithm to define configuration as a mathematical structure, which captures the state of the algorithm. We define also a sequence of configurations, such that every element of this sequence corresponds to the state of the sequential algorithm

*Supported in part by NSF grant CCF-0430695 and by Texas Advanced Research Program Grant 003658-0029-1999.

STOC'05, May 22-24, 2005, Baltimore, Maryland, USA.

at certain points of its execution. We use the sequence of configurations to trivially define an $O(\log^2 n)$ space algorithm, which instead of storing all of its current state, recomputes parts of it when it needs them. This technique is standard for designing space-efficient algorithms. Finally, we modify the $O(\log^2 n)$ space algorithm into an algorithm which uses $O(\log n \log\log n)$ space.

The possibility of simulating parallel algorithms for USTCONN space-efficiently was suggested by Prof. Vijaya Ramachandran in 2000. She conjectured an $O(\log n \log\log n)$ space algorithm derived from the CL algorithm and an alternate simple $O(\log^{3/2} n)$ space algorithm derived from the algorithm of Johnson and Metaxas [11], by using the max-degree hooking scheme of [6]. She observed that the step needing derandomization in [16] is not necessary in a tree-based hook and contract approach, because the trees automatically give rise to disjoint clusters of vertices. The max-degree hooking scheme employed by [6] gives the additional benefit that small trees have small neighborhoods. The main challenge was to implement the levels of recursion, so that they process small trees in $o(\log n)$ space. Solving this problem is the main contribution of this paper.

In the algorithm presented here the space of a level of recursion is between $\Omega(\log\log n)$ and $\Theta(\log n)$, depending on the level. A key tool for our method are the exploration walks on trees defined by Koucký [13]. Exploration walks on trees are similar to the Euler tour technique used by Tarjan and Vishkin [23] in the parallel context. These walks play the role of the edge-list plugging technique and pointer jumping employed by the CL algorithm, because they allow us to traverse trees very efficiently.

Section 2 describes briefly the CL algorithm and the sequential algorithm derived from it. Section 3 gives the formal definition of a labeled multi-graph and operations on graphs, as well as the definition of configuration and sequence of configurations. Section 4 gives an outline of the space-efficient algorithm for solving USTCONN.

2. THE CHONG–LAM ALGORITHM

The Chong–Lam (CL) algorithm [6] uses a hook and contract approach. There are several stages of hooking and contraction. Before every stage every vertex of the original graph is in exactly one of three states – active, inactive, and done; all active and inactive vertices have non-zero degree and there are no multi-edges between active vertices; the inactive vertices are organized in a set of hooking trees.

In a hooking phase the active vertices in parallel choose to hook to one of their current neighbors and thus either become part of existing hooking trees or form new ones. The fact that the components formed by the hooking of vertices are trees is ensured by the hooking scheme of the CL algorithm.

In a contraction phase some of the current hooking trees are contracted to a representative vertex. Which trees are contracted is determined by a parameter, which depends on the phase and sets an upper bound on the total degree, i.e. the sum of the degrees of the vertices, of the trees which are contracted. For every contracted tree its representative becomes a new active vertex and the rest of its vertices become done and are removed from further consideration. Also all multi-edges between new active vertices are cleaned-up. Finally, the vertices of every uncontracted tree become inactive.

The processing required by the hooking is performed in parallel time $O(\log d)$, where d is the degree of the active vertex, using pointer jumping. The important part of the contraction procedure is checking the degree of a hooking tree. In parallel this could be done in $O(\log c)$, where c is the value of the contraction parameter, by using pointer jumping and a constant time edge-list plugging technique.

Finally the CL algorithm is given by the following recursive procedure. Here `MaxHook` and `Contract`(c) denote correspondingly a hooking and a contracting phase with parameter c.

```
procedure Connect(k)
    MaxHook;
    if k > 0 then
        Contract(2^(2^k));
        Connect(k − 1);
        Connect(k − 1);
    Contract(2^(2^(k+1)));
```

The correctness of the CL algorithm ensures that a call to `Connect`($\lceil \log\log n \rceil$) contracts every connected component of the graph to a single vertex and all the other vertices are organized in a set of rooted parent trees such that the root of the tree of a vertex u is the vertex to which the connected component of u contracted.

We simulate the CL algorithm trivially with a sequential algorithm using linear space. We fix an ordering on the edges incident on a vertex and instead of performing the hooking in parallel for all active vertices, we do it sequentially for each of them. This is possible, because by changing the hooking scheme of CL slightly we can ensure that the hooking of an active vertex does not depend on the hooking of the other active vertices. The new hooking strategy gives preference to neighbors which are inactive or have an inactive neighbor, but it still ensures that small trees composed entirely of active vertices have small degree. The details of the sequential algorithm are given in the following section, and the space-efficient versions of the same algorithm are given in section 4.

3. DEFINITIONS

3.1 Graphs, Trees, and Exploration Walks

An undirected multi-graph is a graph with possibly multiple edges between two vertices and such that every edge has a label on each side, where the labels of the edges incident to a vertex v have distinct labels on the side of v. We also have a single self-loop with label 0 at every vertex. Formally we have

Definition 1. An *undirected multi-graph* G is a triple $\langle V, \delta, \mu\rangle$, where V is a set, $\delta : V \to \mathbb{N}$, and $\mu : E \to E$ is a bijection such that $\mu(\mu(e)) = e$ and $\mu(v, 0) = (v, 0)$, where $E = \{(v, i) : v \in V \text{ and } 0 \le i \le \delta(v)\}$.

V is the *set of vertices* of G, E is the *set of edges* of G, $\delta(v)$ is the *degree* of v, and $\mu(e)$ is the *reverse edge* of e. For an edge e, call the set $\{e, \mu(e)\}$ an *undirected edge*.

Let $\eta : E \to V$ and $\beta : E \to \mathbb{N}$ be the first and the second component of μ. Then $\eta(v, i)$ is the *i-th neighbor* of v, i is the *label* of the edge (v, i), and $\beta(v, i)$ is its *back-label*.

Define the *size* of G, size(G), to be $|V|$.

In the following a graph means undirected multi-graph.

Definition 2. A graph $G' = \langle V', \delta', \mu' \rangle$ is a subgraph of G, if $V' \subseteq V$ and for every $u, v \in V'$,

$$|\{i : \eta'(u,i) = v\}| \leq |\{i : \eta(u,i) = v\}|.$$

Define (simple) path, connected vertices, forest and tree in the usual way.

Definition 3. Let G be a graph. Let $\Delta : E \times \mathbb{Z} \to E$ be such that $\Delta((v,i),j)$ changes by j the label of the edge (v,i). More precisely for $i \neq 0$,

$$\Delta((v,i),j) = (v, 1 + (i - 1 + j \bmod \delta(v))).$$

Define $\Gamma_{G,k}, \Gamma'_{G,k} : E \to E$ inductively on $k \geq 0$. First $\Gamma_{G,0}(e) = \Gamma'_{G,0}(e) = e$. Now let

$$\begin{aligned} \Gamma_{G,k+1}(e) &= \Delta(\mu(\Gamma_{G,k}(e)), 1), \\ \Gamma'_{G,k+1}(e) &= \mu(\Delta(\Gamma'_{G,k}(e), -1)). \end{aligned}$$

$\Gamma_{G,k}(e)$ is called an *exploration walk* starting from the edge e. $\Gamma'_{G,k}$ is called *the reverse exploration walk*. Let $e_l = (v_l, i_l) = \Gamma_{G,l}(e)$, for $0 \leq l \leq k$. v_l and e_l are correspondingly the l-th vertex and the l-th edge visited by the exploration walk.

Exploration walks were introduced by Koucký [13]. We define them only for the exploration sequence which always changes by one the label of the current edge because this is the case of interest to us. The fact that exploration walks are reversible, i.e. $\Gamma'_{G,k}(\Gamma_{G,k}(e)) = e$, was noticed by Koucký and is what makes them so important to us. We will use the following universal property of exploration walks on trees.

PROPOSITION 1 ([13]). *Let $e = (v,i) \in E$, $i \neq 0$, and $e_j = (v_j, i_j) = \Gamma_{G,j}(e)$, $j \geq 0$. If G is a tree with at most one undirected edge between any two vertices and $k = 2(\text{size}(G)-1)$, then $e_k = e$ and every edge of G which is not a self-loop appears exactly once in $e_0, \ldots, e_{k-1}$. Furthermore, $2(\text{size}(G)-1)$ is the smallest k such that v appears $\delta(v)+1$ times in $v_0, \ldots, v_k$.*

3.2 Operations on Graphs

3.2.1 Configuration

A configuration is the state of the sequential algorithm outlined at the end of section 2. Formally

Definition 4. A *configuration* is a tuple $\mathcal{C} = \langle G, A, I, D, H, R \rangle$, where G is a graph with $V = [n]$, for some $n \in \mathbb{N}$, and $\delta(v) = 0$, for $v \in D$. A, I, and D form a partition of V. $H : V \to \mathbb{N}$ and $R : V \to V$ are such that $H(v) \leq \delta(v)$ and if $R(v) \neq v$, then $v \in D$.

The elements of A, I, and D are called correspondingly the *active*, the *inactive*, and the *done* vertices of G. For $u \in V$, $(u, H(u))$ is the *hooking edge* of u, and $R(u)$ is the *representative* of u.

We require that H and R do not have non-trivial cycles in the following sense. Let $v_1, \ldots, v_k \in V$, $k \geq 2$. Then

1) if $v_{i+1} = \eta(v_i, H(v_i))$, $i \in [k-1]$, and $v_1 = \eta(v_k, H(v_k))$, then $H(v_1) = 0$, and

2) if $v_{i+1} = R(v_i)$, $i \in [k-1]$, and $v_1 = R(v_k)$, then $R(v_1) = v_1$.

We also require that there is at most one undirected edge between any two active vertices, i.e. if $u, v \in A$, then $|\{i : \eta(v,i) = u\}| \leq 1$, and there is no hooking edge from an inactive to an active vertex.

Define

$$\text{rep}_R(v) = \begin{cases} v, & R(v) = v, \\ \text{rep}_R(R(v)), & \text{otherwise.} \end{cases}$$

By 2) of Definition 4, this definition is correct.

Definition 5. Let $\mathcal{C} = \langle G, A, I, D, H, R \rangle$ be a configuration. H defines a subforest $F = \langle V, \delta_F, \mu_F \rangle$ of G, called the *hooking forest* of $\mathcal{C}$, with at most one undirected edge between any two vertices in the following way.

Fix $v \in V$. Let ε be 1, if $H(v) \neq 0$, and 0, otherwise. Let $0 < i_1 < \cdots < i_k$ be such that

$$\{i_1, \ldots, i_k\} = \{i : \exists u \in V \text{ such that } \mu(u, H(u)) = (v,i)\}.$$

First define $\delta_F(v) = k + \varepsilon$. Now define $\eta_F(v,j) = \eta(v, i_j)$, for $1 \leq j \leq k$, and, if $\varepsilon = 1$, $\eta_F(v, k+\varepsilon) = \eta(v, H(v))$. Finally define $\beta_F(v,j) = i$, where $\eta_F(v,j) = u$ and $\eta_F(u,i) = v$.

Let T be a maximal connected subtree of the forest F. We call T a *hooking tree* in $\mathcal{C}$. The *root* of T, root(T), is the only vertex v in T such that $H(v) = 0$. The *degree* of T, $\deg(T)$, is $\sum_{v \in V(T)} \delta(v)$. For a vertex $v \in V$ we denote with T_v the subtree of F which contains v.

The correctness of this definition and the fact that F is a forest with at most one undirected edge between any two vertices follow from 1) of Definition 4.

3.2.2 Hooking

We will define Hook$(\mathcal{C})$ so that, if $\mathcal{C}$ describes the state of the sequential algorithm, then Hook$(\mathcal{C})$ is its state after one hooking step.

Being elements of $\mathbb{N}$, the elements of V are ordered. Define the linear ordering $<_d$ on V so that

$$u <_d v \text{ iff } \delta(u) < \delta(v) \text{ or } \delta(u) = \delta(v) \text{ and } u < v.$$

The result of the hooking operation Hook$(\mathcal{C})$ is the configuration $\mathcal{C}' = \langle G, A, I, D, H', R \rangle$ defined in the following way. If v is inactive, then $H'(v) = H(v)$. If v is active, let $v_1, \ldots, v_{\delta(v)}$ be the neighbors of v, i.e. $v_i = \eta(v,i)$. For the rest of the definition when we have to choose an index i, we always pick the smallest one with the corresponding property. If v has an inactive neighbor v_i, let $H'(v) = i$. If all neighbors of v are active, let v_i be the largest according to $<_d$ amongst the neighbors of v. If $v <_d v_i$, let $H'(v) = i$. If all neighbors of v are active and smaller than v according to $<_d$, then if v has a neighbor v_i which has an inactive neighbor, let $H'(v) = i$. If all neighbors of v and their neighbors are active, then if v has a neighbor v_i which has a neighbor larger than v according to $<_d$, let $H'(v) = i$. Finally, if all neighbors of v and their neighbors are active and smaller than v according to $<_d$, define $H'(v) = 0$.

The hooking strategy described above differs from the hooking strategy of the CL algorithm because it gives preference to neighbors which are inactive or have an inactive neighbor. For example, in our hooking strategy a vertex hooks to an inactive neighbor, if it has one, regardless of its degree. In the hooking strategy of the CL algorithm, a vertex hooks to its largest according to $<_d$ neighbor, regardless of its state. The new hooking strategy does not change the correctness of the CL algorithm because first an active vertex still declares itself a representative iff all of its neighbors are active and hooked to it, and second along a sequence

of hooking edges of active vertices the vertices increase according to $<_d$ the same way as in the CL algorithm. The reason we chose the new strategy is to ensure that we do not lookup the degree of an inactive vertex.

The fact that Hook($\mathcal{C}$) is a configuration is proven as in [6]. Furthermore, by [6], if T is a hooking tree in Hook($\mathcal{C}$) composed entirely of active vertices, then $\deg(T) \leq \text{size}^2(T)$.

3.2.3 Contraction

We will define Contract($\mathcal{C}, d$) so that, if $\mathcal{C}$ describes the state of the sequential algorithm, then Contract($\mathcal{C}, d$) is its state after one contraction step with parameter d. A hooking tree T in $\mathcal{C}$ is *d-contractable*, if $\deg(T) \leq d$.

The result of Contract($\mathcal{C}, d$) is the configuration $\mathcal{C}' = \langle G', A', I', D', H', R' \rangle$ defined in the following way. First define

$$\begin{aligned} A'' &= \{v : v \notin D \text{ and } \deg(T_v) \leq d \text{ and } \text{root}(T_v) = v\}, \\ I' &= \{v : v \notin D \text{ and } \deg(T_v) > d\}, \\ D'' &= \{v : v \in D \text{ or } \deg(T_v) \leq d \text{ and } \text{root}(T_v) \neq v\}. \end{aligned}$$

Now define

$$H'(v) = \begin{cases} H(v), & v \in I', \\ 0, & \text{otherwise}, \end{cases}$$

and

$$R'(v) = \begin{cases} R(v), & v \in D, \\ \text{root}(T_v), & v \in D'' - D, \\ v, & \text{otherwise}. \end{cases}$$

Let T be a hooking tree in $\mathcal{C}$ and $s = \text{size}(T)$. Let $v_1, \ldots, v_s$ be the enumeration of the vertices of T visited by the exploration walk starting from the edge $(\text{root}(T), 1)$ of T, where we enumerate a vertex only the first time it is visited by the exploration walk. Let $e_1, \ldots, e_k$ be the enumeration of the edges of G incident to the vertices of T defined in the following way – enumerate all edges incident to v_1, then all edges incident to v_2, and so on. Obviously $k = \deg(T)$.

Let us define now G'. For $u \in A''$, define $l_u \in \mathbb{N}$ and the following enumeration of edges $e_{u,j}$, $1 \leq j \leq l_u$. First consider the enumeration $e_1, \ldots, e_k$ of the edges of T_u from the previous paragraph. Remove from this enumeration all edges which are internal to T_u, i.e. such that $\eta(e_i) \in V(T_u)$. From every subsequence of edges whose other end belongs to the same d-contractable hooking tree leave only the first edge, i.e. for every d-contractable hooking tree $T \neq T_u$ of $\mathcal{C}$, if $e_{i_1}, \ldots, e_{i_h}$ are all the edges in the sequence $e_1, \ldots, e_k$ such that $\eta(e_{i_j}) \in V(T)$, leave only e_{i_1}. Let l_u be the number of remaining edges in the enumeration and $e_{u,j}$, $1 \leq j \leq l_u$, be their enumeration. Naturally we call the edges $e_{u,j}$, the *remaining edges of* T_u.

Define

$$\begin{aligned} A' &= A'' - \{v \in A'' : l_v = 0\}, \\ D' &= D'' \cup \{v \in A'' : l_v = 0\}, \end{aligned}$$

and

$$\delta'(v) = \begin{cases} l_v, & v \in A', \\ \delta(v), & v \in I', \\ 0, & v \in D'. \end{cases}$$

We are left to define $\mu'(v, i)$. First assume $v \in A'$. Let $(u, j) = \mu(e_{v,i})$. If T_u is not d-contractable, then define $\mu'(v, i) = (u, j)$. If T_u is d-contractable, then define $\mu'(v, i) = (w, k)$, where $w = \text{root}(T_u)$ and k is the only index such that $\eta(e_{w,k}) \in V(T_v)$. Now assume $v \in I'$. Let $(u, j) = \mu(v, i)$. If T_u is not d-contractable, then define $\mu'(v, i) = (u, j)$. If T_u is d-contractable, let $\mu'(v, i) = (w, k)$, where $w = \text{root}(T_u)$ and k is the only index such that $e_{w,k} = (u, j)$.

From the definition of $\mathcal{C}' = \text{Contract}(\mathcal{C}, d)$ follows that $\mathcal{C}'$ is a configuration such that $\delta'(v) \leq d$, for $v \in A'$, and $\deg(T_v) > d$, for $v \in I'$. Furthermore, in the hooking forest F' every $v \in A' \cup D'$ is in a hooking tree which contains only v.

3.3 Sequence of Configurations

For every $k \in \mathbb{N}$, we will define a sequence of configurations $\mathcal{C}_l$, $0 \leq l \leq r(k)$.

First, define recursively a rooted tree C_k (here a tree is used in the usual sense) of depth $k + 1$ whose leaves are labeled. The root of C_0 has two leaf descendants – the left labeled with $(0, 0)$ and the right with $(1, 1)$. The root of C_k, $k \geq 1$, has four descendants. From left to right they are: first a leaf labeled with $(0, k)$, second a leaf labeled with $(1, k)$, then two copies of C_{k-1}, and finally a leaf labeled with $(1, k + 1)$. By induction it can be proven easily that C_k has $r(k) = 5 \cdot 2^k - 3$ leaves. Number starting from 1 the leaves of C_k as they appear in its left to right traversal.

For $0 \leq l \leq r(k)$, let $\sigma(k, l)$ be $(0, k)$, if $l = 0$, and the label of the l-th leaf of C_k, otherwise. Let $\sigma_{1,k}(l)$ and $\sigma_{2,k}(l)$ be correspondingly the first and the second component of $\sigma(k, l)$. In the following we omit k from the subscripts of $\sigma_{1,k}$ and $\sigma_{2,k}$.

Let $\mathcal{C}_0$ be some configuration. Assume that we have already defined $\mathcal{C}_0, \ldots, \mathcal{C}_{l-1}$ for $1 \leq l \leq r(k)$. Define

$$\mathcal{C}_l = \begin{cases} \text{Hook}(\mathcal{C}_{l-1}), & \sigma_1(l) = 0, \\ \text{Contract}(\mathcal{C}_{l-1}, 2^{2^{\sigma_2(l)}}), & \text{otherwise}. \end{cases}$$

Let C'_k be the labeled tree, which is obtained from C_k by substituting the label of the l-th leaf with `MaxHook`, if $\sigma_1(l) = 0$, and `Contract`$(2^{2^{\sigma_2(l)}})$, otherwise. The structure of C'_k follows the structure of the recursive calls of `Connect`(k) – the levels of C'_k are the levels of recursion of `Connect`(k). Thus the configurations in the sequence defined in the previous paragraph are exactly the states of the sequential algorithm after consecutive hooking and contraction operations, where $\mathcal{C}_0$ is its state initialized according to the input graph.

Define a configuration $\mathcal{C}_l$ to be *nice*, if 1) $\text{size}(T_v) > 2^{2^{h+1}}$ and $\deg(T_v) > 2^{2^{h+2}}$, for $v \in I_l$, and 2) $\delta_l(v) \leq 2^{2^{h+2}}$, for $v \in A_l$, where $h = \sigma_2(l) - \sigma_1(l)$.

By definition, $\mathcal{C}_l$ is nice iff the state described by it fulfills the preconditions given in [6] for executing `Connect`(h). These preconditions ensure the correctness of the algorithm and that it can be executed efficiently in parallel.

Considering the correspondence between the sequence $\mathcal{C}_0$, $\ldots, \mathcal{C}_{r(k)}$ and the `Connect`(k) procedure of the sequential algorithm, the following theorem is a consequence of the results in [6].

THEOREM 1. *If $\mathcal{C}_0$ is a nice configuration, then $\mathcal{C}_l$ is nice, for every $1 \leq l \leq r(k)$, $|A_{r(k)}| \leq \max\{|A_0|/2^{2^k}, 1\}$, and $\text{size}(T_v) > 2^{2^{k+1}}$, for $v \in I_{r(k)}$.*

Finally, again by [6], we have the following corollary, which

says that if we initialize $\mathcal{C}_0$ according to some undirected graph G, then in $\mathcal{C}_r$ all components of G are contracted.

COROLLARY 1. *Let G be a graph with at most one undirected edge between any two vertices and $V = [n]$. Let $k = \lceil \log \log n \rceil$ and $\mathcal{C}_0 = \langle G, A, I, D, H, R \rangle$, where $A = \{v : \delta(v) \neq 0\}$, $I = \emptyset$, $D = V - A$, $H(v) = 0$ and $R(v) = v$, for $v \in V$. Then u and v are connected in G iff $\mathrm{rep}_{R_{r(k)}}(u) = \mathrm{rep}_{R_{r(k)}}(v)$.*

4. SPACE-EFFICIENT ALGORITHM

The algorithms described in sections 4.1 and 4.2 are most precisely given in pseudo-code. Due to a lack of space we can only give an outline of their definition here. Precise definitions could be found in [24]. We use fixed-width font to denote the names from the pseudo-code. The pseudo-code is executed on a Turing Machine which has one of its tapes designated to serve as the stack of execution. The stack contains the local variables of the functions executed at the moment and is used in the usual way – a stack frame is allocated on the top of the stack at the beginning of the execution of a function and released at its end. The stack frame also contains the address in the program from which the function was called. The algorithms also use global variables (each global variables is on a separate tape of the Turing Machine) of which there are at most a constant number and each takes space $O(\log n)$.

The space of the algorithms described in sections 4.1 and 4.2 is dominated by their execution stacks. Both algorithms are recursive with $O(\log n)$ levels of recursion, where the level of recursion never increases at a function call. Also, there are at most a constant number of nested calls within the same level of recursion. Thus we have a stack frame for each level of recursion which contains the stack frames of the functions executed at this level, and which has a total of at most a constant number of local variables. For the algorithm described in section 4.1, all local variables take $\Theta(\log n)$ space, and for the algorithm described in section 4.2 their space depends on the level of recursion. More precisely, all local variables at level l are at most $v(l)$ for some function v which we will specify later. We call a numerical value *valid* for level l, if it is at most $v(l)$. The function $v(l)$ is easily computable in $O(\log n)$ space. We devote a separate tape for the computation of $v(l)$, so that the space necessary to compute it does not appear in the space taken by the execution stack.

Arguments to functions and return values are implemented through global variables straightforwardly. For all functions from section 4.1 and most from section 4.2 this is sufficient. For some functions in section 4.2 this is not enough and we introduce a new method of passing arguments and returning values which uses global arrays. We describe this method next.

Let `F` be a function which uses this method of passing arguments and returning values. We have two global arrays – one, `argF`, for passing arguments to `F` and one, `retF`, for returning a value from it. Those arrays contain exactly one entry for every possible level of recursion and each entry could be marked. Also each entry holds values which are valid for the corresponding level of recursion.

Let `H` calls `F`. If `H` uses the value returned by `F`, then before the call to `F` the entry of `retF` for the current level of recursion is marked, otherwise it is left unmarked. When `F` produces a result, it finds in `retF` the first marked entry after the entry for the current level of recursion and tries to store its result there. If the value produced is too large for the corresponding entry, `F` writes `null`. After the call to `F` returns, `H` unmarks the entry of `retF` for the current level of recursion.

Similarly, if `H` provides arguments to `F`, then before the call to `F`, `H` marks the entry of `argF` for the current level of recursion and provides values for it. When `F` wants to access its arguments it looks up, starting from the current level, and finds the first entry of `argF` which is marked and uses the values stored in the entry as its arguments. After the call to `F` returns, `H` unmarks the entry of `argF` for the current level of recursion.

This method can be used, if `F` is always called on the previous level of recursion, i.e. all calls to `F` are $\texttt{F}(l-1,\ldots)$, because then there is no danger of overwriting its arguments or returned value. For passing arguments this method helps when we need to store an argument only at the level which generates it (where it is valid), but still allow for lower levels of recursion to access it (where it might be invalid). For returning values this method helps when we want to return a value exactly at the level which requested it. This method allows for nested calls from different levels of recursion to the same function, which the ordinary method of passing variables through global variables does not always allow

In section 4.1 we outline an $O(\log^2 n)$ space implementation of the sequential algorithm derived from the definitions in sections 3.2 and 3.3, which instead of storing all of the current configuration, recomputes parts of it when it needs them. In section 4.2 we describe the changes we make to the algorithm from section 4.1 to reduce its space complexity to $O(\log n \log \log n)$.

4.1 An $O(\log^2 n)$ Space Algorithm

Let G be a graph with $V = [n]$ and with at most one undirected edge between any two vertices. Let $r = 5 \cdot 2^{\lceil \log \log n \rceil} - 3$.

Consider the sequence of configurations $\mathcal{C}_0, \ldots, \mathcal{C}_r$ from Corollary 1. The starting point for a space-efficient algorithm comes directly from the definition of this sequence. More precisely, we define functions $\texttt{Active}(l, v)$, $\texttt{Inactive}(l, v)$, $\texttt{Done}(l, v)$, $\texttt{Degree}(l, v)$, $\texttt{Neighbor}(l, v, i)$, $\texttt{BackLabel}(l, v, i)$, $\texttt{Hook}(l, v)$, and $\texttt{Rep}(l, v)$, where $0 \leq l \leq r$, v is a vertex, and i is the label of an edge incident to v, which return the corresponding component of $\mathcal{C}_l$.

Call the value of the parameter l, the *level of recursion*. Thus the levels of recursion of our algorithm correspond to the elements of the sequence $\mathcal{C}_0, \ldots, \mathcal{C}_r$. For $l = 0$ all of these functions just use the input graph G (this is the bottom of the recursion), and their output for level $l + 1$ is determined from outputs for level l according to the definitions in section 3.3. Using these functions, to solve undirected st-connectivity we apply Corollary 1.

If $\sigma_1(\lceil \log \log n \rceil, l) = 1$, then $\mathcal{C}_l$ is obtained from $\mathcal{C}_{l-1}$ by contracting some of its hooking trees as defined in section 3.2.3. In this case for a hooking tree T in $\mathcal{C}_{l-1}$ we must determine $\deg(T)$ and be able to enumerate the vertices of T as they are visited by the exploration walk on T starting from $(v, 1)$, for $v \in V(T)$. For these purposes we define $\texttt{TreeSize}(l, v)$ and $\texttt{TreeWalk}(l, v, i)$. Let T be the hooking tree in $\mathcal{C}_{l-1}$ containing v. If s is the size of T, $\texttt{TreeSize}(l, v)$ returns $2(s-1)$. Notice that $2(s-1)$ is the length of the ex-

ploration walk on T given in Proposition 1. `TreeWalk`(l, v, i) returns $\Gamma_{T,i}(v, 1)$.

The following claim can be proven by going over the details of the definitions of each of the functions mentioned above. It is not hard to see that each level of recursion takes $O(\log n)$ space.

CLAIM 1. *The functions* `Active`(l, v), `Inactive`(l, v), `Done`(l, v), `Degree`(l, v), `Neighbor`(l, v, i), `BackLabel`(l, v, i), `Hook`(l, v), *and* `Rep`(l, v), *correctly return the corresponding components of* $\mathcal{C}_l$. *The total space taken by the execution of each of these functions is* $O(l \log n)$.

4.2 The $O(\log n \log\log n)$ Space Algorithm

The $O(\log n)$ space per level in Claim 1 comes mainly from having to store vertices in the local variables of the functions, since each vertex takes $\Theta(\log n)$ space. To take care of this bottleneck we define the functions so that they never keep a vertex in their local variables.

The first step towards such definitions is to remove the vertex v from the argument list of the functions. Instead of this argument, we maintain one current vertex in a global variable and all functions return information about this vertex. A function which otherwise must return a vertex is defined so that after its execution the current vertex is its result (in this case we say that the function moves the current vertex). It is a responsibility of the calling function to keep enough information locally to restore the original current vertex, if it needs to. Denote the current vertex with cv.

To implement this, first we change some of our functions. Instead of `Neighbor`(l, v, i), we have `Neighbor`(l, i), which moves the current vertex to its i-th neighbor in $\mathcal{C}_l$. Let T be the hooking tree of cv in $\mathcal{C}_{l-1}$. Instead of `TreeWalk`(l, v, i) we have `TreeForward`(l, i), which returns j and moves the current vertex to u, where $(u, j) = \Gamma_{T,i}(cv, 1)$. Similarly we have `TreeBack`$(l, i, j)$ which moves the current vertex to the end vertex of $\Gamma'_{T,i}(cv, j)$.

The most important part of our idea to avoid storing vertices locally is to be able to move the current vertex temporarily, perform something at the new current vertex, and then return to the original current vertex. For this define `Move`(l, i) to return $\beta_{l-1}(cv, i)$ and move the current vertex to $\eta_{l-1}(cv, i)$. Call `Move`(l, i) and `TreeForward`(l, i) *forward moves*. For a forward move M, let `Reverse`(M, j) be its reverse, i.e. it is correspondingly `Move`(l, j) or `TreeBack`(l, i, j), where j is the result of M. We use the reversibility of exploration walks here, so that `TreeBack` reverts `TreeForward`.

We use forward moves to change the current vertex and their reverses to restore it. Call a sequence of forward moves *path description relative to the current vertex*. If `P` is a path description relative to the current vertex and `B` is some instruction(s), then define **after** `P` **do** `B` to change the current vertex according to `P`, perform `B`, and then use the reverses of the moves in `P` to restore the current vertex.

A simple example of the use of `after` is the comparison operator `=`, which compares two vertices given their path descriptions relative to the current vertex and returns true iff they are the same. Using `after` we can move to the first vertex and store it in a local variable, then go to the second vertex and compare the two. This takes $\Theta(\log n)$ space. Instead of this, going back and forth between the two vertices, using the reversibility of the moves along the edges and the exploration walks on the trees, we perform the comparison bit by bit. Aside from the information stored for the ways back, this takes only the $\Theta(\log\log n)$ space necessary to store the index of a bit. This way the bottleneck of $\Omega(\log n)$ space is reduced to $\Omega(\log\log n)$.

For example, we use the `=` operator in the definition of `TreeSize` in the following way. Using Proposition 1, we can make steps from the exploration walk on a hooking tree T in $\mathcal{C}_{l-1}$ until we go back to the starting vertex v sufficiently many times. For this we can store v, so that we can compare it with each new vertex of the walk. This takes $\Theta(\log n)$ space, independent of the degree of T. Instead, we incrementally find i with properties as in Proposition 1, where to check if the i-th vertex of the walk is equal to v, we keep the current vertex at v and use the `=` operator, as defined above, to compare it to the vertex with path description `TreeForward`(l, i). Thus, if $\deg(T) = d$, then we can find its size in space $O(\max\{\log d, \log\log n\})$.

The second part of our idea to reduce the space of the algorithm is to have an upper bound $v(l)$ on the values which variables can take at level l, i.e. during the execution of all functions at level l the values of their local variables are at most $v(l)$. We set $v(l) = 2{\cdot}2^{2^{k+2}}$, where $k = \sigma_2(\lceil\log\log n\rceil, l)$ (see section 3.3). Call a number x *valid* for level l, if it is at most $v(l)$. A vertex v is *valid* for level l, if its degree $\delta_{l-1}(v)$ is valid for level l.

Using the concept of a current vertex, we can eliminate the need to store a vertex in a local variable and thus our local variables contain only degrees of vertices, indices of neighbors, back-labels of edges, and lengths of exploration walks on hooking trees. For example, the information stored for reversing a sequence of forward moves are back-labels (for `Move`) and tree-edge labels (for `TreeForward`). We still have to make sure that every time we store a value in a local variable it is valid. For this the following observation is helpful.

OBSERVATION 1. *1) The labels of the edges incident to vertex v valid for level l are valid for level l. This is not necessarily true for their back-labels. 2) All vertices which are active in $\mathcal{C}_{l-1}$ are valid for level l. 3) If a hooking tree T in $\mathcal{C}_{l-1}$ is contractable for level l, then all of its vertices are valid for level l.*

The first item of the observation is trivial, the second follows from Theorem 1, because all $\mathcal{C}_l$ are nice, and the third follows because $v(l)$ is bigger than the contraction parameter for level l.

Our goal has become to prove the following lemma. Let T be the hooking tree in $\mathcal{C}_{l-1}$ of the current vertex cv. Let (cv, i) be an edge of T and $v = \eta_T(cv, i)$. We call a move along (cv, i) *possible for level* l, if v is valid for level l. T is *contractable for level* l, if $\deg(T) \leq d$, where d is the contraction parameter for level l, i.e. $d = 2^{2^{\sigma_2(\lceil\log\log n\rceil, l)}}$.

LEMMA 1. *1.* `Active`(l), `Inactive`(l), `Done`(l), `Hook`(l), *and* `Degree`(l), *correctly return the value of the corresponding component of* $\mathcal{C}_l$ *for* cv.

2. If T is uncontractable for level l or $cv \in D_{l-1}$, then `Root`(l) *returns 0, otherwise it returns the index of the first occurrence of* $\mathrm{root}(T)$ *in the exploration walk on T starting from* $(cv, 1)$.

`TreeSize`(l) *returns* $2($`size`$(T)-1)$, *if T is contractable for level l, and* `null` *otherwise.*

Assume that cv is valid for level l. If all moves of $\Gamma_{T,i}(cv, 1)$ are possible for level l and it ends in (v, j), then `TreeForward`(l, i) *moves the current vertex to v and returns j. If all moves of $\Gamma'_{T,i}(cv, j)$ are possible for level l and it ends in v, then* `TreeBack`(l, i, j) *moves the current vertex to v.*

3. *Let $(v, j) = \mu_l(cv, i)$.* `Neighbor`(l, i) *moves the current vertex to v;* `BackLabel`(l, i) *returns j, if j is valid for the level at which* `BackLabel`(l, i) *was called, and* `null` *otherwise.*

All local variables are valid.

The proof of the lemma is done by induction on the level of recursion. For this we need the correctness of the functions which a given function calls. Sometimes we have to use correctness for the same level of recursion, but this does not result in a circular reasoning because for any two functions `F` and `G`, there are no chains of function calls within the same level of recursion both from `F` to `G` and from `G` to `F`.

The correctness essentially follows from the correctness of the CL algorithm (Corollary 1) – the only change is to give priority to inactive vertices in the hooking operation, as described in section 3.2.2, but this does not change the overall correctness of the algorithm. It can be easily seen that the introduction of one global current vertex and always returning information about this vertex, maintains the faithfulness of our implementation to the CL algorithm – the current vertex is an implicit argument to all functions describing a configuration and calling it "current" just facilitates our intuition about how the algorithm proceeds.

The only real deviation from the definitions given in section 3.2 is that we have an upper bound on the numerical values for a level, and so we might be unable to process the result of a function, if it is invalid for the current level of recursion. Actually, as can be seen from Lemma 1, some functions are specified to return `null`, if their result is invalid for the level requesting it. The only information we can derive from an invalid or `null` result is that either the current vertex or a neighbor of the current vertex is invalid, or that the current vertex is part of an uncontractable tree. This information is enough to define the functions as in Lemma 1. First, we never move to a vertex from which we cannot return (i.e. along an edge with an invalid back-label), so we never have to store an invalid back-label locally. Second, vertices which are either invalid or part of an uncontractable tree are inactive and thus, by definition, inherit their properties from the previous level. In a hooking operation (section 3.2.2), we do not need to lookup the degree of an invalid (and even inactive) vertex. In a contraction operation (section 3.3.3), we can stop the exploration walk on a tree as soon as the walk runs into an invalid vertex, because then the tree is clearly uncontractable.

To ensure that all local variables are valid for the current level of recursion we use Observation 1 in the following way. First, notice that since the value returned from a function resides in a global variable we can check whether it is valid by simply inspecting this global variable. Furthermore, some functions (e.g. `TreeSize` and `BackLabel`) return `null`, if their result is invalid for the current level of recursion. In any case, we can easily learn when the return value of a function is invalid without having to store it locally. According to 1) of Observation 1, if the result of `BackLabel` is invalid, then the corresponding neighbor is invalid and we never move the current vertex along such edges. Also, 2) of Observation 1 ensures that we can process locally active vertices. Finally, according to 3) of Observation 1, if the result of `TreeSize` is invalid, then the hooking tree of the current vertex is uncontractable.

4.3 Solving Undirected st-Connectivity

Using Lemma 1 we can prove the main theorem of this paper.

THEOREM 2. *USTCONN on a graph with n vertices can be solved in space $O(\log n \log\log n)$.*

By Corollary 1, s and t are connected iff $\mathrm{rep}_{R_r}(s) = \mathrm{rep}_{R_r}(t)$. Thus to solve USTCONN it is enough to define a function which moves the current vertex cv to $\mathrm{rep}_{R_l}(cv)$.

Let $m = \lceil \log\log n \rceil$. The space complexity of the algorithm is dominated by the space taken by the execution stack. From Lemma 1 follows that each local variable at level l is at most $v(l)$. Since there are constant number of local variables per function and the length of every chain of function calls within the same level of recursion is bounded by a constant, the space taken by level l is $O(\max\{\log v(l), m\})$ (the additional $O(m)$ space appears because of comparison of vertices). Since $\log v(l) = O(2^{k(l)})$, $1 \le l \le r$, where $k(l) = \sigma_2(m, l)$, we have to prove that

$$\sum_{l=1}^{r} \max\{2^{k(l)}, m\} = O(\log n \log\log n).$$

Consider the recurrence given by

$$S(k) = \begin{cases} 2m, & k = 0 \\ 2S(k-1) + 3\max\{2^k, m\}, & k > 0. \end{cases}$$

From the recursive definition of C_r given in section 3.3 follows that the left hand side of the equation which we want to prove is exactly $S(m)$. Finally it is not hard to prove by induction that $S(m) = O(\log n \log\log n)$.

Acknowledgments The author is grateful to Prof. Anna Gál for help in the preparation of this paper and Prof. Vijaya Ramachandran for pointing out the problem and many helpful discussions.

5. REFERENCES

[1] R. Aleliunas, R. Karp, R. Lipton, L. Lovász, and C. Rackoff. Random walks, universal traversal sequences, and the complexity of maze problems. In *20th IEEE Symposium on Foundations of Computer Science*, pages 218–223, 1979.

[2] R. Armoni, A. Ta-Shma, A. Wigderson, and S. Zhou. $SL \subseteq L^{\frac{4}{3}}$. In *20th ACM Symposium on Theory of Computing*, pages 230–239, 1997.

[3] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for ultracomputer and PRAM. In *International Conference on Parallel Processing*, pages 175–179, 1983.

[4] F. Chin, J. Lam, and I.-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25:659–666, 1982.

[5] K. Chong, Y. Han, and T. Lam. On the parallel time complexity of undirected connectivity and minimum spanning trees. *Journal of the ACM*, 48(2):297–323, 2001.

[6] K. Chong and T. Lam. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. *Journal of Algorithms*, 18(3):378–402, 1995.

[7] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. In *27th IEEE Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[8] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. In *27th IEEE Symposium on Foundations of Computer Science*, pages 492–501, 1986.

[9] S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems. In *7th ACM-SIAM Symposium on Discrete Algorithms*, pages 438–447, 1996.

[10] D. Hirschberg, A. Chandra, and D. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[11] D. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} |V|)$ parallel time for the CREW PRAM. In *32nd IEEE Symposium on Foundations of Computer Science*, pages 688–697, 1991.

[12] D. Karger, N. Nisan, and M. Parnas. Fast connected components algorithm for the EREW PRAM. In *4th ACM Symposium on Parallel Algorithms and Architectures*, pages 373–381, 1992.

[13] M. Koucký. Universal traversal sequences with backtracking. In *16th IEEE Conference on Computational Complexity*, pages 21–27, 2001.

[14] H. Lewis and C. Papadimitriou. Symmetric space-bounded computation. *Theoretical Computer Science*, 19:161–187, 1982.

[15] N. Nisan. Psuedorandom generators for space-bounded computation. In *22nd ACM Symposium on Theory of Computing*, pages 204–212, 1990.

[16] N. Nisan, E. Szemeredi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *33rd IEEE Symposium on Foundations of Computer Science*, pages 24–29, 1992.

[17] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM Journal on Computing*, 31(6):1879–1895, 2002.

[18] O. Reingold. Undirected st-connectivity in log-space. In these proceedings, 2005.

[19] C. Savage and J. JáJá. Fast, efficient parallel algorithms for some graph problems. *SIAM Journal on Computing*, 10(4):682–691, 1981.

[20] W. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[21] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[22] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[23] R. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985.

[24] V. Trifonov. An $O(\log n \log\log n)$ space algorithm for undirected st-connectivity. Technical Report TR04-114, Electronic Colloquium on Computational Complexity, http://www.eccc.uni-trier.de/eccc/, 2004.