An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM

Shay Halperin *   *Uri Zwick* *

October 17, 1994
revised: October 30, 1995 and May 8, 1996

Abstract

Improving a long chain of works we obtain a randomised EREW PRAM algorithm for finding the connected components of a graph $G = (V, E)$ with n vertices and m edges in $O(\log n)$ time using an optimal number of $O((m + n)/\log n)$ processors. The result returned by the algorithm is always correct. The probability that the algorithm will not complete in $O(\log n)$ time is $o(n^{-c})$ for any $c > 0$.

1 Introduction

Finding the connected components of an undirected graph is perhaps the most basic algorithmic graph problem. While the problem is trivial in the sequential setting, it seems that elaborate methods should be used to solve the problem efficiently in the parallel setting. A considerable number of researchers investigated the complexity of the problem in various parallel models including, in particular, various members of the PRAM family. In this work we consider the EREW PRAM model, the weakest member of this family, and obtain, for the first time, a parallel connectivity algorithm that achieves the minimal possible running time using the minimal possible number of processors. The algorithm we obtain is randomised.

Relatively simple CRCW PRAM algorithms that find the connected components of a graph $G = (V, E)$ deterministically in $O(\log n)$ time using $m + n$ processors were obtained by Shiloach and Vishkin [SV82] and by Awerbuch and Shiloch [AS87]. More complicated deterministic CRCW PRAM connectivity algorithms that run in $O(\log n)$ time using $O((m+n)\alpha(m, n)/\log n)$ processors, where $\alpha(m, n)$ is a functional inverse of the Ackermann function, were obtained by Cole and Vishkin [CV91] and Iwama and Kambayashi [IK94]. The number of processors can be reduced to the optimal number of $O((m + n)/\log n)$ if randomisation is allowed (Gazit [Gaz91]). Until not long ago, the best CREW PRAM connectivity algorithm used $O(\log^2 n)$ time (Hirschberg, Chandra and Sarwate [HCS79]) (Chin, Lam and Chen [CLC82]). The problem of designing an $o(\log^2 n)$ time CREW PRAM algorithm was open for over a decade (cf. [KR90]) until a deterministic $O(\log^{1.5} n)$ time CREW PRAM algorithm that uses $m + n$ processors was obtained by Johnson and Metaxas [JM91]. Johnson and Metaxas have later shown [JM92] that their algorithm can also be implemented in the EREW PRAM model. At about the same time, Karger, Nisan and Parnas [KNP92] used the interesting technique of short random walks on graphs, developed initially by Aleliunas, Karp, Lipton, Lovasz and Rackoff [AKL+79], to develop a randomised EREW PRAM algorithm that runs in either $O(\log n)$ time using $O((n^{1+\epsilon} + m)/\log n)$ processors, for any $\epsilon > 0$, or in $O(\log n \log\log n)$ time using

*Department of Computer Science, School of Mathematical Sciences, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv 69978, Israel. E-mail addresses : {hshay,zwick}@math.tau.ac.il.
Work supported in part by **The basic research foundation** administrated by **The Israel academy of sciences and humanities**.

$O((m+n)/\log n)$ processors. Their algorithm can also be derandomised, using universal sequences, yielding a deterministic algorithm that matches the performance of the algorithm of Johnson and Metaxas. Finally, Chong and Lam [CL95] obtained a deterministic EREW PRAM algorithm that runs in $O(\log n \log\log n)$ time using $m+n$ processors. For a comparison of some of these and some other algorithms from a practical point of view see Greiner [Gre94].

Concurrently and independently of our work, Radzik [Rad94] had recently obtained a randomised EREW PRAM connectivity algorithm that runs in $O(\log n)$ time using $m+n$ processors.

In this work we combine methods from many of the previous works to obtain a randomised EREW PRAM algorithm that runs in $O(\log n)$ time using $O((m+n)/\log n)$ processors. A running time of $O(\log n)$ is best possible in the EREW PRAM model (Cook, Dwork and Reischuk [CDR86], Dietzfelbinger, M. Kutyłowski and Reischuk [DKR94]), even if randomisation is allowed, and $\Omega((m+n)/\log n)$ processors are clearly necessary to obtain a running time of $O(\log n)$. Our result is therefore optimal.

Sparse graphs usually pose the greatest difficulty to algorithms for finding connected components. Note, for example, that the algorithm of Cole and Vishkin [CV91] is optimal if $m=\Omega(n\log^* n)$ and that the algorithm of Karger, Nisan and Parnas [KNP92] is optimal if $m=\Omega(n^{1+\epsilon})$ for some $\epsilon>0$. Our algorithm is unusual in the sense that it *reduces* the problem of finding the connected components of a graph $G=(V,E)$ to the problem of finding the connected components of a sparse graph $G'=(V',E')$ with $O(m+n)$ vertices and edges. Every vertex in G' has, however, a degree of at most three. By carefully controlling the growth of the maximal degree of the graph G' at the various stages of the algorithm we are able to obtain our optimal algorithm.

Fast and efficient parallel algorithms for special classes of sparse graphs, such as planar graphs, were developed by Hagerup [Hag90]. Hagerup obtained in particular a deterministic $O(\log n \log^* n)$ time optimal speedup EREW PRAM algorithm for finding the connected components of planar graphs.

An overview of our algorithm is given in the next section. One simple concept that should be defined beforehand is the following: A graph $G'=(V',E')$ is an *image* of a graph $G=(V,E)$ if there exists a mapping $h:V\rightarrow V'$ such that for every $u,v\in V$, u and v are in the same connected component of G if and only if $h(u)$ and $h(v)$ are in the same connected component of G'. The mapping h is called the *image function*. The problem of finding the connected components of a graph $G=(V,E)$ is equivalent to the problem of finding an image $G'=(V',E')$ of $G=(V,E)$ in which every vertex is isolated together, of course, with the image function $h:V\rightarrow V'$. A graph $G'=(V',E')$ is said to be a *partial image* of a graph $G=(V,E)$ if there exists a mapping $h:V\rightarrow V'$ such that for every $u,v\in V$, if $h(u)$ and $h(v)$ are in the same connected component of G' then u and v are in the same connected component of G. In the sequel when we say that an algorithm obtains an image, or a partial image of a graph, we mean that this algorithm outputs the image, or partial image, as well as a corresponding image function.

One of the tools used to obtain our algorithm is a new load balancing scheme that may be of interest in its own right. It is described in a general setting in Section 4.

In the next section we give an overview of our algorithm. The simple idea used to transform every graph into a graph with maximal degree at most three is described in Section 3. Although this idea is completely trivial, it forms one of the bases of our algorithm. In Section 4 we describe our new load balancing scheme which is used by our algorithm. As shall be explained in the next Section, our connectivity algorithm is composed of two main stages. The first stage of our algorithm is described in Part I of the paper comprising Sections 5 to 9. The second stage of our algorithm is described in Part II of the paper comprising Sections 10 to 16. We then end in Section 17 with some concluding remarks and open problems. A preliminary version of this paper had appeared in [HZ94].

2 An overview of the algorithm

The input to the connectivity algorithm is a graph $G = (V, E)$ with n vertices and m edges. The graph is specified using its adjacency lists. Our algorithm is composed of two main stages. The first stage takes the input graph $G = (V, E)$ and produces an image $G' = (V', E')$ of it in which $|E'| = O((m+n)/\log^2 n)$. Isolated vertices in G' represent complete connected components of G and can be removed. The graph G' will now contain at most $O((m+n)/\log^2 n)$ vertices. The second stage takes the reduced graph and finds its connected components. Both stages take $O(\log n)$ time using $O((m+n)/\log n)$ processors. The failure probability of the first stage is at most $2^{-\Omega(2^{\log n/(\log\log n)^2})}$. The failure probability of the second stage is at most $2^{-\Omega(2^{\sqrt{\log n}})}$. Both these probabilities are $o(n^{-c})$ for any $c > 0$. In the rest of the paper, when we say that an event occurs with *very high probability*, we mean that the probability that it does not occur is $O(n^{-c})$, for some fixed constant $c > 0$. By appropriately setting some relevant parameters, we can usually make the constant c arbitrarily large.

The first stage (the size reduction) uses an adaptation of a method developed by Gazit [Gaz91] to reduce the number of non-isolated vertices, followed by an application of a random sampling method Karger [Kar93] (used also by Karger *et al.* [KNP92]) to reduce the number of edges. The method of Gazit was developed for the CRCW PRAM model and adapting it to run in the EREW PRAM model required subtle changes.

The second stage (finding the connected components) uses an adaptation of the random walk method of Karger, Nisan and Parnas [KNP92]. Their algorithm finds the connected components of a graph $G = (V, E)$ in $O(\log n)$ time using $O((n^{1+\epsilon} + m)/\log n)$ processors, for any fixed $\epsilon > 0$. Our version of their algorithm runs in $O(\log n)$ time using only $O((m+n)\log n)$ processors (or even $O(m+n)$ processors, as explained in Section 16). This enables us to find the connected components of the graph $G' = (V', E')$, in which $|V'|, |E'| = O((m+n)/\log^2 n)$, in $O(\log n)$ time using $O((m+n)/\log n)$ processors, as required. To improve the algorithm of Karger, Nisan and Parnas [KNP92] we use some ideas inspired by the growth control techniques of Johnson and Metaxas [JM91] and Chong and Lam [CL95].

The algorithm of Radzik [Rad94] also uses the random walk method of Karger, Nisan and Parnas [KNP92] and there are several similarities between it and the second stage of our algorithm.

The first stage of our connectivity algorithm receives a graph $G = (V, E)$ and produces an image $G^{(1)} = (V^{(1)}, E^{(1)})$ of it, along with an image function $h^{(1)} : V \to V'$. The second stage receives the graph $G^{(1)} = (V^{(1)}, E^{(1)})$ and produces an image $G^{(2)} = (V^{(2)}, \phi)$ of it, along with an image function $h^{(2)} : V^{(1)} \to V^{(2)}$. To complete the operation of the whole algorithm, we should compose the mapping $h^{(1)}$ and $h^{(2)}$ and obtain the composed mapping $h = h^{(1)} \circ h^{(2)}$. Such a composition can be easily implemented in $O(1)$ time if concurrent reads are allowed. To enable a quick and efficient implementation of this composition without concurrent reads, the first and second stages also find the inverses $(h^{(1)})^{-1}$ and $(h^{(2)})^{-1}$ of the image functions $h^{(1)}$ and $h^{(2)}$. For every vertex $v' \in V'$, $(h^{(1)})^{-1}(v')$ is a linked list that contains all the vertices of V that are mapped by $h^{(1)}$ into v'. Using the inverses $(h^{(1)})^{-1}$ and $(h^{(2)})^{-1}$ it is easy to compute the composition $h = h' \circ h''$ in $O(\log n)$ time using $O(n/\log n)$ processors in the EREW PRAM model.

Our implementation of Gazit's size reduction algorithm [Gaz91] and our improvement of the algorithm of Karger, Nisan and Parnas [KNP92] are made possible by carefully controlling the degrees of the vertices in the various stages of the algorithm. Both the first and the second stages begin by obtaining an image of the graph given to them whose maximal degree is at most three. This gives us, in both cases, favourable starting points. We then make sure that the degrees of the vertices of the intermediate graphs constructed by our algorithm do not grow too fast. The simple way used to obtain the images with maximal degree three is described in the next section. A second essential ingredient for the implementation of the size reduction stage (the first stage) of our algorithm is the simple load balancing scheme described in Section 4.

3 Obtaining an image with maximal degree three

Both stages of our algorithm begin by taking a graph $G = (V, E)$ with n edges and m edges, throwing away its isolated vertices, and obtaining an image $G' = (V', E')$ of it with $O(m)$ vertices and edges and maximal degree at most three. Isolated vertices can be thrown away as they constitute complete connected components. The method used to obtain the image $G' = (V', E')$ is extremely simple. Each vertex v of G of degree $k > 3$ is replaced by a path (or a cycle) of length k. Each one of the k edges adjacent to v is then connected to one of the vertices of this path. The obtained graph is clearly an image of G and its maximal degree is at most three.

We shortly sketch the implementation of this simple transformation. We assume that the graph $G = (V, E)$ is specified using its adjacency lists and that each edge $(u, v) \in E$ has a pointer to its opposite edge $(v, u) \in E$. Using an optimal list-ranking algorithm (Anderson and Miller [AM91]) we give serial numbers to the edges of each adjacency list. A vertex v of G of degree k is replaced in G' by k vertices $v_1, \ldots, v_k$ that form a path (or a cycle). If (u, v) and (v, u) are edges of the G (which is undirected) and (u, v) is the i-th edge in the adjacency list of u and (v, u) is the j-th edge in the adjacency list of v, then an edge (u_i, v_j) is added to G'. The function that assigns to each vertex v in G the vertex v_1 in G' is the image function from G to G'. The graph G' contains $n' = 2m$ vertices and $m' \leq 1.5n' = 3m$ edges. Using an optimal prefix-sums algorithm (Ladner and Fischer [LF80]) it is easy to construct, in $O(\log n)$ time using $O((m + n)/\log n)$ processors, an $n' \times 3$ matrix that lists for each vertex of G' its (up to three) neighbours.

4 A load balancing scheme

To solve the processor allocation problem in the size reduction stage of our algorithm we use a simple load balancing scheme. As this scheme may also be useful in other situations, we describe this scheme here in a fairly general setting.

Other load balancing schemes were obtained by Cole and Vishkin [CV88], for the EREW PRAM, and by Gil, Matias and Vishkin [GMV91], Goodrich [Goo91] and Hagerup [Hag92],[Hag93], for the CRCW PRAM. These balancing schemes are much more sophisticated than our balancing scheme. Yet, neither one of them suits our purposes.

Let ALG be a (possibly randomised) loosely specified[1] parallel algorithm that uses m virtual processors. The execution of ALG is composed of K phases. The i-th phase takes $t_i \geq 1$ time units and at most m_i of the virtual processors are active in it. Assume, at first, that the set of active processors at the $i + 1$-st phase is a subset of the set of processors active at the i-th phase. At the end of each phase, each virtual processor that participated in that phase knows whether it should stay active during the next phase or whether it should become and stay idle during all subsequent phases. We do not assume anything else about the sets of active processors.

The total time required by the loosely specified algorithm is clearly $T = \sum_{i=1}^{K} t_i$. The total amount of work performed by the active processors is $W = \sum_{i=1}^{K} m_i t_i$. We would like to obtain an EREW PRAM implementation of this algorithm using P processors whose total running time is as close as possible to $O(W/P + T)$. That is, we want an implementation of Brent's scheduling principle [Bre74] (see also JáJá [JáJ92], p. 28) that does not ignore the processors allocation issue. We assume that $P, T \leq m$ and that $W/P = \Omega(\log m)$.

We divide the actual processors into P/b groups of size $b = \log m$ (we assume for simplicity that b is an integer dividing P). Each such group is initially assigned with mb/P of the virtual processors. To record the allocation of the virtual processors to the actual processors, we use a $P/b \times mb/P$ matrix

[1] The term *loosely specified* is used in [Goo91] and [GMV91] in a broader sense than that used here. Our loosely specified algorithms are similar to the *task decaying* algorithms of [Goo91] and [GMV91].

MAT. The virtual processors allocated to the i-th group are listed in the i-th row of this matrix. Before we start the simulation of the loosely specified algorithm, we apply an independent random cyclic rotation on each one of the mb/P columns of MAT. The amount by which each column is rotated is chosen uniformly at random from $\{0, 1, \ldots, P/b-1\}$. These random rotations can be easily implemented on an EREW PRAM in $O(m/P+\log m)$ time using P processors. Perhaps surprisingly, this simple method guarantees, as we shall see, a relatively balanced partition of the work among the processors throughout the course of the algorithm.

Each phase of ALG is simulated in the following way. Each group of b actual processors equally divides the virtual processors allocated to the group among its b members. Let L_i be an upper bound on the number of virtual processors allocated to a single actual processor before starting to simulate the i-th phase of ALG. The i-th phase of ALG can be easily simulated then in $O(L_i t_i)$ time. After the simulation of the i-th phase is completed, each group of b processors compresses the list of virtual processors allocated to it removing from it all the virtual processors that became idle. Once the active processors are compressed, it is easy to divide them equally among the processors of the group. Such a compression of a list whose size is at most $L_i b$ using b processors can be easily implemented on an EREW PRAM in $O(L_i + \log b) = O(L_i + \log\log m)$ time. The total amount of time required for the simulation of the i-th phase is therefore $O(L_i t_i + \log\log m)$.

Let $A > 3$ be a fixed constant. We claim that with a probability of at least $1 - m^{-\Omega(A)}$, the number of virtual processors allocated to each actual processor during the simulation of the i-th phase of ALG is at most $L_i = A \cdot (\frac{m_i}{P} + 1)$. Equivalently, the number of virtual processors allocated to each group of $b = \log m$ processors is at most $\bar{L}_i = bL_i = A \cdot (\frac{m_i b}{P} + b)$, again with a probability of at least $1 - m^{-\Omega(A)}$.

As ALG may be a randomised algorithm, the exact set of active processors at the i-th phase may be a random variable. We show that for every possible set S_i of at most m_i active processors at the i-th phase, the probability that a certain group of b actual processors is assigned with more than $\bar{L}_i$ processors at the i-th phase is at most $m^{-\Omega(A)}$. Multiplying this bound by PK/b, we get a bound on the probability that any group of processors will get too much work during any phase of the algorithm. As $PK/b = m^{O(1)}$, the obtained bound is still of the form $m^{-\Omega(A)}$.

To obtain this probability bound we use one of the standard Chernoff bounds (see Alon and Spencer [AS92], p. 237). Let $X_1, X_2, \ldots, X_n$ be mutually independent random variables, not necessarily identically distributed, that take only the values 0 and 1. Let $X = \sum_{i=1}^{n} X_i$ and let $\mu = E[X]$. Then, for every $\beta \geq 1$ we have $\Pr[X \geq \beta\mu] \leq (e^{\beta-1}/\beta^{\beta})^{\mu}$. A simple manipulation shows that for any $M \geq \mu$ we have $\Pr[X \geq M] \leq (e\mu/M)^M$.

Suppose that S_i is the set of active virtual processors at the i-th phase of ALG. Consider a specific group of b actual processors. Let X_j be a random variable which is 1 if the j-th virtual processor originally allocated to the group is active in the i-th phase of the algorithm, and 0 otherwise. Then, $X = \sum_{j=1}^{mb/P} X_j$ is the number of active processors allocated to the group at the i-th phase. As a random rotation was applied on the j-th column of MAT, the j-th processor allocated to the group was chosen uniformly at random among the processors of the j-th column of MAT. Thus $E[X_j] = \Pr[X_j = 1] = m_{ij}/(P/b)$, where m_{ij} is the number of processors in the j-th column of MAT that are active at the i-th phase (i.e., members of S_i). So, $\mu = E[X] = \sum_j m_{ij}/(P/b) \leq m_i b/P$. The random rotations performed on the columns of MAT are mutually independent. The X_j's are therefore also mutually independent. As $\bar{L}_i \geq A\mu$ and $\bar{L}_i \geq A \log m$, we may apply the Chernoff bound and obtain that $\Pr[X \geq \bar{L}_i] \leq (e\mu/\bar{L}_i)^{\bar{L}_i} \leq (e/A)^{A\log m}$. As $A > 3$, this is indeed of the form $m^{-\Omega(A)}$ as promised (it is even of the form $m^{-\Omega(A\log A)}$).

The simulation of the i-th phase takes therefore, with very high probability, at most $O((\frac{m_i}{P} + 1)t_i + \log\log m)$ time. The total time required for the simulation of ALG is therefore $O(\sum_{i=1}^{K}(\frac{m_i}{P} + 1)t_i + K\log\log m) = O(W/P + T + K\log\log m)$. To this we should add the $O(\log m)$ time required for the initial random rotations. This term is gobbled up by the $O(W/P)$ term as we assume that

$W/P = \Omega(\log m)$. Comparing this to the expression we hoped for, we see that we got an extra $O(K \log\log m)$ term. Our simulation is therefore optimal whenever $K = O(\log m/\log\log m)$.

In the situation we encounter in the first stage of our algorithm we have $W = O(m)$, $P = O(m/\log m)$, $K = O((\log\log m)^2)$ and $T = O((\log\log m)^3)$. The extra $O(K \log\log m)$ term is therefore negligible and the total running time will be $O(\log m)$.

The choice $b = \log m$ made above is the minimal choice possible. By choosing a larger value of b, the time required for the compression after each phase increases but the probability of failure decreases. The optimal choice for b is in fact $b = 2^{(\frac{W}{P}+T)/K}$, assuming that this number is at least $\log m$. The time required for all the compression operations is then $K \log b = O(\frac{W}{P}+T)$ and the failure probability is only $2^{-\Omega(b)}$. In the case of the first stage of our algorithm, we can take $b = 2^{\log m/(\log\log m)^2}$ and the failure probability is then at most $2^{-\Omega(2^{\log m/(\log\log m)^2})}$, as promised.

To allow the specification of the size reduction algorithm of the next section in the framework set in this section, we have to allow the loosely specified algorithm a bit more freedom. The loosely specified algorithm is allowed to keep a stack to which the configurations of the active processors in each phase are pushed. The algorithm is then allowed to return to a previously used configuration, reviving some of the virtual processors if necessary. It is easy to check that the simulation result obtained holds also for this more general class of loosely specified algorithms. This is done by keeping a stack with the matrices *MAT* used during the simulation.

PART I : SIZE REDUCTION

5 Reducing the size of the graph

In this section we describe the skeleton of the size reduction stage of our algorithm. This stage receives an input graph $G = (V, E)$ with n vertices and m edges and produces an image $G'' = (V'', E'')$ of it with at most $O((m+n)/\log^2 n)$ edges and non-isolated vertices.

The size reduction is performed in two substages. In the first we obtain an image $G' = (V', E')$ of G with at most $O((m+n)/\log^4 n)$ non-isolated vertices. This is done using an adaptation of a method of Gazit [Gaz91]. We then obtain an image $G'' = (V'', E'')$ of G' with at most $O((m+n)/\log^4 n)$ non-isolated vertices and $O((m+n)/\log^2 n)$ edges. This is done using a simple sampling idea of Karger [Kar93], also used by Karger, Nisan and Parnas [KNP92], whose analysis was slightly strengthened recently by Klein and Tarjan [KT94].

The input graph $G = (V, E)$, without its isolated vertices, is initially transformed, as described in Section 3, into a graph with $O(m + n)$ vertices and edges and with maximal degree at most three. We assume therefore, for simplicity, that the input graph $G = (V, E)$ is already of this form, i.e., that the maximal degree of G is at most three and that therefore $m \le 1.5n$.

This section is broken into three subsections. In the first we give a high level description of our algorithm. In the second we mention some of the implementation details of this stage. The different procedures used in the first stage, and their detailed implementations, are described in the subsequent sections. In the third subsection we analyse the complexity of the algorithm.

5.1 High level description

The size of the graph $G = (V, E)$ is reduced by running the procedure *REDUCE* whose description is given in Figure 1. The procedure *REDUCE* calls the procedures *COUPLE*, *DILUTE*, *IMAGE* and *SPARSIFY*. The specifications of these procedures are given in Figure 2.

REDUCE(G_0)

input: A graph $G_0 = (V, E)$ with $|V| = n$ vertices, $|E| \leq 1.5n$ edges and maximal degree 3.

output: An image $G'' = (V'', E'')$ of G_0 with at most $O(n/\log^2 n)$ edges and non-isolated vertices.

complexity: $O(\log n)$ time using $O(n/\log n)$ processors.

```
for i ← 0 to k − 1 (where k = O(log log n)) do
begin
{
    G'_i ← COUPLE(G_i)
    G_{i+1} ← DILUTE(G'_i)
}
end
G' ← IMAGE(G_0, G_k)
G'' ← SPARSIFY(G')
return(G'')
```

Figure 1: The procedure *REDUCE*.

The procedure *REDUCE* employs the general strategy used by Gazit [Gaz91]. The implementation details however are completely different. *REDUCE* begins by performing $O(\log\log n)$ phases. In each phase a call is made to the procedure *COUPLE* that reduces, with a very high probably, the number of non-isolated vertices in the graph by a factor of at least $\beta = 12/13$. The procedure *COUPLE* achieves this by first finding a large enough 'matching' in the graph. It then contracts each 'edge' of this matching. The words 'matching' and 'edges' are quoted as the matching found is actually a matching in the square of the graph. Each 'edge' in this matching may actually be a path of length two in the current graph. The reduction of the number of non-isolated vertices by a constant factor achieved by *COUPLE* does not imply a similar reduction in the number of edges. To obtain a linear work algorithm we have to *dilute* the edges of the graph at each phase. This is done by the procedure *DILUTE* that makes sure that the number of edges in the graph is at most a constant factor times the number of non-isolated vertices in the graph. To achieve this, *DILUTE* deletes certain edges from the graph. This may cause certain connected components of the graph to break into several smaller connected components. The number of such new connected components formed by *DILUTE* is relatively small however.

The graph obtained after these $O(\log\log n)$ phases has a small enough number of non-isolated vertices but is only a partial image of G. The procedure *IMAGE* is then applied to produce an image of the original graph with the same set of vertices. This may slightly increase the number of non-isolated vertices and drastically increase the number of edges. Finally, a call to the procedure *SPARSIFY* is made. This call reduces the number of edges by the required amount.

Before describing the procedures *COUPLE*, *DILUTE*, *IMAGE* and *SPARSIFY* in more detail, we show that *REDUCE* does fulfill its task, assuming that these four procedures meet the specifications given in Figure 2. The procedures *COUPLE*, *DILUTE* and *SPARSIFY* are randomised and they may fail to achieve their goals but this will only happen with a very small probability ($o(n^{-c})$ for any $c > 0$).

Let n_i be the number of non-isolated vertices in G_i, let m_i be the number of edges in G_i and let d_i be the maximal degree of G_i (refer to the pseudo-code given in Figure 1). At the start we have $n_0 = n$, $m_0 \leq 1.5n$ and $d_0 \leq 3$. As each call to *COUPLE* reduces the number of non-isolated vertices by a

COUPLE(G_i)

input: A graph $G_i = (V_i, E_i)$ with n_i non-isolated vertices, m_i edges and maximal degree at most d_i.

output: An image $G_i' = (V_i', E_i')$ of G_i with at most βn_i non-isolated vertices and maximal degree at most $2d_i$.

complexity: $O(\log d_i)$ time (in a single phase) and $O(m_i \log d_i)$ work using $m_i + n_i$ virtual processors.

DILUTE(G_i)

input: A graph $G_i = (V_i, E_i)$ with n_i non-isolated vertices, m_i edges and maximal degree at most d_i.

output: A subgraph $G_i' = (V_i, E_i')$ of G_i such that $|E_i'| \le A \cdot n_i$ and such that G_i' contains at most $O(n_i/\log^4 n)$ new connected components.

complexity: $O((\log d_i) \cdot \log\log n + (\log\log n)^2)$ time (in $O(\log\log n)$ phases) and $O(m_i \log d_i)$ work using $m_i + n_i$ virtual processors.

IMAGE(G, G')

input: A graph $G = (V, E)$, with $|V| = n$ and $|E| = m$, and a partial image $G' = (V', E')$ of it.

output: An image $G'' = (V', E'')$ of G, i.e., an image of G with the vertex set V'.

complexity: $O(\log n)$ time using $O((m+n)/\log n)$ processors.

SPARSIFY(G)

input: A graph $G = (V, E)$ with $|V| = n$ vertices and $|E| = m$ edges.

output: An image $G' = (V', E')$ of G with at most $O(n \log^2 n)$ edges.

complexity: $O(\log n)$ time using $O((m+n)/\log n)$ processors.

Figure 2: The specifications of *COUPLE*, *DILUTE*, *IMAGE* and *SPARSIFY*.

factor of β, we get that $n_i \le \beta^i n$. A call to *COUPLE* at most doubles the maximal degree. Thus $d_i \le 3 \cdot 2^i$. The call to *DILUTE* ensures that $m_i \le A n_i$, for some fixed $A > 0$, and that G_i contains at most $O(n_i/\log^4 n)$ new connected components. The number of non-isolated vertices in G_k is at most $O(n/\log^4 n)$, provided that $k \ge 4 \log_{1/\beta} \log n$.

As the transition from G_i to G_{i+1} creates at most $O(n_i/\log^4 n)$ new connected components, the number of connected components in G_k that do not correspond to original connected components of G_0 is $O(\sum_{i=0}^{k-1} n_i/\log^4 n)$, which is $O(n/\log^4 n)$, as the n_i's form a decreasing geometric sequence. Each isolated vertex in G_k that is not an isolated vertex in G' is such a new connected component. The number of non-isolated vertices in G' is therefore also $O(n/\log^4 n)$. The call to *SPARSIFY* now generates an image G'' of G_0 with at most $O(n/\log^2 n)$ edges, as required from the output of *REDUCE*.

5.2 Some implementation details

The input to *REDUCE* is a graph $G_0 = (V_0, E_0)$ with n vertices, $m \leq 1.5n$ edges, and maximal degree at most three. It is represented using an $n \times 3$ matrix N_0. Each row of N_0 corresponds to a vertex v of G_0. The row $N_0[v]$ contains the up to three edges adjacent to v in G_0.

The procedure *REDUCE* constructs a sequence of graphs $G_0 = (V_0, E_0), G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)$, where $k = O(\log \log n)$. The number of vertices in G_i is, with very high probability, at most $n_i \leq \beta^i n$, where $\beta = 12/13$. The maximal degree of G_i is at most $d_i = 3 \cdot 2^i$. The vertex sets $V_0, V_1, \ldots, V_k$ of $G_0, G_1, \ldots, G_k$ form a decreasing sequence $V_0 \supseteq V_1 \supseteq \ldots \supseteq V_k$ of sets. Each vertex of one of these graphs is also a vertex therefore of the original graph G_0. The graph G_i is represented using an $n \times d_i$ matrix N_i. If v is a vertex of G_i, and $u_1, \ldots, u_r$, where $r \leq d_i$, are the neighbours of v in G_i, then the row $N_i[v]$ contains the edges $(v, u_1), (v, u_2), \ldots, (v, u_r)$ in some order. The entry that holds the edge (v, u_j) also contains a pointer to the entry in $N_i[u_j]$ that holds the edge (u_j, v). If v is not a vertex of G_i then the row $N_i[v]$ is not in use.

The matrix N_i used to hold G_i is of size $n \times d_i$, and not $n_i \times d_i$ as might been expected, as we cannot afford to compress the rows of the matrices N_i after each iteration of *REDUCE*. This would consume too much time. The first stage, as described here, requires therefore slightly more than linear, namely $n \log^{O(1)} n$, space. A copy of N_0 should be kept for the use of *IMAGE*. The copy of N_i, for $i > 0$, may be destroyed however once N_{i+1} has been constructed. The space requirements of the first stage can be reduced by using methods similar to those used in the load balancing scheme of Section 4. As the paper is long enough as it is, and as the space requirements of the second stage of our algorithm are larger than those of the first stage, we do not pursue this matter any further.

The description of *REDUCE*, *COUPLE* and *DILUTE* is greatly simplied by assuming that each edge and each non-isolated vertex of G_i has a virtual processor allocated to it. As a vertex becomes isolated and as an edge is deleted from the graph, the virtual processor associated with it becomes idle. This fits the general framework of the previous section. The execution of *DILUTE* involves a certain backtracking process that uses the stack mechanism. The complexity bounds of *COUPLE* and *DILUTE* specified in Figure 2 are given in this loosely specified setting.

The transition from G_i to G_{i+1} involves, among other things, the deletion of edges and the contraction of pairs of vertices. When edges are deleted, each row of the array N_i should be compressed. This can be done in $O(\log d_i)$ time using the virtual processors allocated to the edges of G_i and in particular to the used elements of N_i. Similarly, when a pair of vertices is contracted, the row of N_i that corresponds to one of them should be appended to the row of the second, internal edges, if there are any, should be removed and the identity of the endpoints of all the edges should be updated. This can again be done in $O(\log d_i)$ time using the virtual processors allocated to the edges of G_i. A closer look at these operations shows that they can be implemented in $O(\log d_i)$ time without the use of any concurrent reads or writes. This is of course essential as our algorithm is to run on an EREW PRAM. More detailed implementation details will appear in the subsequent sections of the paper.

The parallel time of almost all the operations performed by our algorithm is determined by the maximal degree of the graph. This is the reason for carefully controlling the degrees of the graphs constructed.

5.3 Complexity analysis

We now bound the complexity of *REDUCE*. Consider at first the complexity of the `for` loop in the idealised setting, assuming that a virtual processor is initially allocated to every vertex and edge of G_0. The total time taken by the for loop is then

$$O(\sum_{i=0}^{k-1}[(\log d_i)\cdot\log\log n + (\log\log n)^2]) = O(\sum_{i=0}^{k-1}[i\cdot\log\log n + (\log\log n)^2]) = O((\log\log n)^3)$$

in $O((\log\log n)^2)$ phases, and as $\sum_{i\geq 0} i\beta^i = O(1)$, the total work is

$$O(\sum_{i=0}^{k-1} m_i \log d_i) = O(n \cdot \sum_{i=0}^{k-1} i\beta^i) = O(n).$$

Using the load balancing scheme of the previous section the for loop can be simulated, with very high probability, in $O(\log n)$ time using $O(n/\log n)$ processors. The failure probability of this simulation is at most $2^{-\Omega(2^{\log n/(\log\log n)^2})}$ which is $o(n^{-c})$ for any $c > 0$. The failure probability of *REDUCE* due to other reasons is exponentially small.

The procedure *IMAGE* again requires $O(\log n)$ time using $O(n/\log n)$ processors. The call to *SPARSIFY* also takes only $O(\log n)$ time using $O(n/\log n)$ processors. The total running time of *REDUCE* is therefore $O(\log n)$ time using $O(n/\log n)$ processors.

6 The procedure *COUPLE*

The input to *COUPLE* is a graph $G = (V, E)$ with n non-isolated vertices and maximal degree at most d. The graph is specified using an $n_0 \times d$ matrix as explained in Section 5.2, where n_0 is the number of vertices of the original graph handed to *REDUCE*. We assume again that a virtual processor is allocated to each non-isolated vertex and each edge of G.

To reduce the number of non-isolated vertices in G by a constant factor we find a matching in the square of G whose size is, with very high probability, at least $n/13$. Each pair in this matching is connected by either a single edge or a path of length two. We note that the paths of length two that connect matched vertices need not be disjoint.

We then contract the pairs of vertices that appear in this matching and obtain, with very high probability, a graph with at most $12n/13 = \beta n$ vertices and maximal degree at most $2d$, as required.

Assume, for simplicity, that no vertex in V is isolated. For every $v \in V$ we let $f(v)$ be a random neighbour of v. The graph $G_f = (V, E_f)$, where $E_f = \{(v, f(v)) : v \in V\}$ is a *pseudo-forest*, i.e., a directed graph in which the outdegree of each vertex is one. Let $Sib \subset V$ be the set of vertices v for which the indegree of $f(v)$ in G_f is at least two. The vertices of Sib are divided into disjoint sets of siblings, where u and v are siblings if and only if $f(u) = f(v)$. The size of each such set is at least two and at most d. Within a set of $k \geq 2$ siblings, we can define a matching of size at least $(k-1)/2 \geq k/3$. This gives us a matching whose size is at least $|Sib|/3$ between the elements of Sib.

We now remove the elements of Sib from G_f. The graph G_f is now composed of isolated vertices, paths and cycles. Let *Iso* be the set of isolated vertices, let *Pth* be the set of vertices contained in paths (of length at least two) and let *Cyc* be the set of vertices contained in cycles (of length at least two). We now apply the *random mating* method of Reif [Rei84] (see also [MR89]). Each vertex $v \in Pth \cup Cyc$ chooses, with equal probabilities, a random colour $colour(v)$ from $\{0, 1\}$. If $colour(v) = 0$ and $colour(f(v)) = 1$, assuming that $f(v)$ exists, we add $(v, f(v))$ to the matching. On each path or cycle of size $k \geq 2$ there are at least $(k-1)/2 \geq k/3$ disjoint pairs. Each such pair joins the matching, independently, with a probability of $1/4$. Finally, note that $|Iso| \leq |Sib|$, as each element in *Iso* points to an element in Sib and no two elements in *Iso* point to the same element in Sib. The expected size of the matching that will be obtained is therefore at least $n/12$. Using the Chernoff bounds it is easy to show that the probability that the size of the matching will be less than, say, $n/13$ is exponentially small.

The method used to construct the matching admits many variations. It is not difficult to modify it to obtain a smaller value of β. The method presented was chosen for its simplicity.

The method described uses randomisation for two different purposes. The first use of randomisation is in the construction of the pseudo-forest. The second use is in the application of the random mating method of Reif [Rei84]. The use of randomisation in the construction of the pseudo-forest may not seem essential at this point. Indeed, the procedure will meet all the requirements put forth so far even if, say, the first edge leaving each vertex will be chosen for the pseudo-forest. The procedure *COUPLE* is invoked however not only from *REDUCE*, but also from *DILUTE* (as will be described in the next section). It is essential for the correctness of *DILUTE* that the pseudo-forests used by *COUPLE* are chosen at random. The use of the random mating method of Reif [Rei84] may be replaced by extracts from the *deterministic mating* method of Gazit [Gaz91] but this leads to a more complicated algorithm.

6.1 The implementation of *COUPLE*

The procedure *COUPLE* gets the $n_0 \times d$ array N that holds the adjacency lists of the graph $G = (V, E)$. We assume for simplicity that no vertex of G is isolated. A virtual processor is therefore allocated to every vertex and edge of G. The procedure *COUPLE* also gets an array h of size n_0. The array h is used to hold the matching constructed by *COUPLE*. Initially $h(v) = v$ for every $v \in V$. The operation of *COUPLE* is composed of the following steps:

Pseudo-forest construction:

Each vertex $v \in V$ chooses, with equal probabilities, one of the edges adjacent to v, i.e., one of the elements of the row $N[v]$. Let $f(v)$ be the other end of the edge chosen.

Finding siblings:

Each vertex $v \in V$ marks the edge $(f(v), v)$ in $N[f(v)]$ as an *incoming* edge. This is done using the pointer from the edge $(v, f(v))$ in $N[v]$ to the edge $(f(v), v)$ in $N[f(v)]$. Each vertex $v \in V$ constructs a list of its incoming edges. This is done by first copying $N[v]$ to a temporary storage location and then compressing this list, leaving in it only the edges marked as incoming.

Matching siblings:

If a vertex $v \in V$ has at least two incoming edges, all the vertices pointing to it are marked as *siblings*. These markings are performed by the processors allocated to the edges of v. These processors also hook, i.e., match, consecutive pairs of siblings. If $u_1, \ldots, u_k$ are the vertices pointing to v then $h(u_1) \leftarrow u_2, \ldots, h(u_{2\lfloor k/2 \rfloor - 1}) \leftarrow u_{2\lfloor k/2 \rfloor}$. The edges $(u_1, v), \ldots, (u_{2\lfloor k/2 \rfloor}, v)$ are marked as *connectors*. The connector markings are used by *DILUTE*.

Random mating:

Every vertex $v \in V$ that was not marked as a sibling chooses a random colour $colour(v) \in \{0, 1\}$ with equal probabilities. If v has only one incoming edge (u, v) in the pseudo-forest G_f, and if $colour(u) = 0$ and $colour(v) = 1$, then v is hooked on u, i.e., $h(u) \leftarrow v$, and the edge (u, v) is marked as a connector.

Contractions:

If $h(v) \neq v$, then all the appearances of v as one of the endpoints of an edge are changed to $h(v)$. The edges emanating from v are kept in $N[v]$ and are easily updated. The edges that enter v are easily accessed using the pointers of the edges of $N[v]$. The whole operation takes only $O(\log d)$ parallel time. Next, the list $N[v]$ is appended to the list $N[h[v]]$ and the appropriate pointers are updated. This takes only $O(\log d)$ time. Self-loops are now marked for deletion and then deleted by compressing the adjacency lists. This again takes only $O(\log d)$ time.

The contractions may create parallel edges. We could, if we wanted too, remove these parallel edges by sorting the edges in each adjacency list and then remove the duplicates. This can be implemented in $O(\log d)$ time using, e.g., Cole's parallel merge sort [Col88]. However, it is essential for the correctness of *DILUTE* that parallel edges are *not* removed by *COUPLE*.

It is easy to see that all these steps can be implemented without concurrent reads or writes in $O(\log d)$ time, assuming that a virtual processor is allocated to each non-isolated vertex and each edge of the graph. The total work is then $O(m \log d)$, where m is the number of edges in the graph.

The same h array is handed to *COUPLE* whenever it is called from *REDUCE*. After the `for` loop of *REDUCE*, the array h defines a forest. The height of each tree in this forest is at most $k = O(\log\log n)$. The vertices in each tree of this forest are exactly the vertices contracted into the root of this tree. Each root of this forest is a vertex of G_k. Using a method described in Subsection 7.1 we compute an array h^* of size n such that for every $v \in V = V_0$, $h^*(v) \in V_k$ is the root of the tree containing v. We also construct for every vertex $v' \in V_k$ a list $(h^*)^{-1}(v')$ that contains the vertices that belong to the tree whose root is v'. The functions $h^* : V_0 \to V_k$ and $(h^*)^{-1} : V_k \to 2^{V_0}$ are the image and inverse image functions from G_0 to G_k.

7 The procedure *DILUTE*

The procedure *DILUTE* receives a graph $G = (V, E)$ with n non-isolated vertices, m edges and maximal degree at most d. The procedure *DILUTE* should delete edges from G thus obtaining a subgraph $G' = (V, E')$ of it in which the number of edges is at most An, for some fixed $A > 0$. The number of new connected components formed by these deletions should be at most $O(n/\log^4 n_0)$, where n_0 is the number of vertices in the original graph.

A *cluster* C of a graph $G = (V, E)$ is a set of vertices that belong to the same connected component of G. Note that we do not require the graph induced by C to be connected. The procedure *DILUTE* attempts to identify a small set of clusters in G whose elements are 'responsible' for a substantial part of the edges in the graph. Such clusters will be called *extrovert clusters*. Edges connecting vertices in two different extrovert clusters are then removed from the graph. This follows the approach of Gazit [Gaz91].

The operation of *DILUTE* is somewhat similar to the operation of *REDUCE*. The details, and especially the correctness proof, are much more complicated however. *DILUTE* also performs $k = \lceil 4 \log_{1/\beta} \log n_0 \rceil = O(\log\log n_0)$ iterations, where $\beta = 12/13$. Each iteration involves a sampling stage in which some of the edges are thrown away and another stage in which vertices of low degree are removed.

In addition to the constant $\beta = 12/13$, *DILUTE* also uses the constants $\alpha = 99/100$ and $\Delta = 100$. It is easy to verify that $\alpha^4 > \beta$ and that $\Delta > \alpha/(1-\alpha)$. A complete description of *DILUTE* is given in Figure 3. The implementation details will be described in Subsection 7.1. The correctness of *DILUTE* follows from the following two Theorems.

Theorem 7.1 *The graph $G' = (V, E')$ obtained by DILUTE is a subgraph of $G = (V, E)$ and it has, with very high probability, at most $O(n/\log^4 n_0)$ connected components which are not connected components of G.*

DILUTE(G_0)
`for` $i \leftarrow 0$ `to` $k-1$ (where $k = O(\log\log n)$) `do`
{

Sample edges:

Each edge of G_i stays in G_i with probability $\alpha = 99/100$ or is removed from G_i with probability $1-\alpha$. All decisions are made independently. (The sampling is performed to obtain a geometrical decrease in the number of edges considered in each iteration, thereby obtaining a linear work algorithm.)

Mark introvert vertices:

Mark all the vertices of G_i whose degree is less than $\Delta = 100$ as *introvert*.

Contract:

Activate *COUPLE* on the graph G_i with the following modifications. Introvert vertices do not choose edges emanating from them to the pseudo-forest (edges going into introvert vertices may be chosen to be in the pseudo-forest). A pair of matched vertices is contracted only if both vertices are not introvert. Each edge of G_i, for $0 \le i \le k$, is an image of an edge in G. During all the above operations, each edge of G_i retains a pointer to its source in G. Whenever an edge in G_i is marked by *COUPLE* as a connector, the same mark is copied to its source in G.

Throw away introvert vertices:

Throw away from G_i all the introvert vertices and all the edges adjacent to them. Each pair of vertices of the matching constructed by *COUPLE* is either contracted, or at least one of its vertices is eliminated. Let G_{i+1} be the new graph obtained. The number of vertices in G_{i+1} is, with very high probability, at most βn_i.

}

Build stars:

The calls to *COUPLE* produce an array h. For every $v \in V$, if $h(v) \neq v$ then v was contracted in some iteration into $h(v)$. The array h defines a forest. The depth of this forest is at most $k = O(\log\log n)$. Using a method described in Subsection 7.1, compute for each vertex $v \in V$, the root $h^*(v)$ of the tree containing v.

Mark vertices:

Let V_k be the set of vertices of the graph G_k obtained after all the iterations. Each vertex $v \in V$ for which $h^*(v) \in V_k$ is marked as an *extrovert vertex*. Each vertex for which $h^*(v) \notin V_k$ is marked as an *introvert vertex*.

Mark edges:

Each edge $(u,v) \in E$ for which $h^*(u) = h^*(v)$ is marked as an *internal* edge. Each edge $(u,v) \in E$ for which $h^*(u) \neq h^*(v)$ and both $h^*(u)$ and $h^*(v)$ are extrovert is marked as an *extrovert edge*. Each edge $(u,v) \in E$ for which $h^*(u) \neq h^*(v)$ for which at least one of $h^*(u)$ and $h^*(v)$ is introvert is marked as an *introvert edge*.

Throw away internal and extrovert edges:

Each extrovert or internal edge which is not a connector is removed from G. Let $G' = (V, E')$ be the graph obtained.

output:

Output the graph $G' = (V, E')$.

Figure 3: The procedure *DILUTE*.

Proof: The function $h^* : V \to V$ computed by *DILUTE* defines a forest of stars. The vertices of each star form a cluster of G. The stars whose centres are vertices of V_k are the extrovert clusters and their vertices are the extrovert vertices of G. Let $A_1, \ldots, A_{n'}$ be the extrovert clusters. As *DILUTE* includes $k = \lceil 4 \log_{1/\beta} \log n_0 \rceil$ calls to *COUPLE* we get, with very high probability, that $n' \leq n / \log^4 n_0$.

The graph G' is obtained from G by removing the internal and the extrovert edges that are not connectors. Let $Con \subseteq E$ be the set edges marked as connectors during the run of *DILUTE*. An internal edge is an edge that connects two vertices that belong to the same cluster. An extrovert edge is an edge that connects a vertex in some extrovert cluster A_i with a vertex in some other extrovert cluster A_j. Let $Int \subseteq E$ be the set of internal edges and let $Ext \subseteq E$ be the set of extrovert edges in G. The edge set of the graph $G' = (V, E')$ is then $E' = (E - Int - Ext) \cup Con$. The removal of internal edges that are not connectors does not create any new connected components as the connectors supply an alternative path for each internal edge removed.

Consider now the effect of the removal of the extrovert edges that are not connectors. We claim that the connected components in G' that contain the extrovert clusters $A_1, \ldots, A_{n'}$ are the only connected components in $G' = (V, E')$ that may not be connected components of G. As each A_i is also a cluster in G' there can be at most $n' \leq n / \log^4 n_0$ such new connected components.

Consider a connected component C of G. If C does not contain any extrovert vertices then only internal edges are removed from C and C is also a connected component of G'. Suppose therefore that C does contain extrovert vertices. Let $v \in C$. Consider a simple path p from v to some extrovert vertex $u \in C$ that does not go through any intermediate extrovert vertices. None of the edges of p is extrovert thus v is also connected to u in G'. Thus, every vertex v of G' is either contained in a connected component which is also a connected component of G, or it is contained in a possibly new connected component that contains an extrovert vertex. This completes the proof. $\square$

Theorem 7.2 *The number of edges in $G' = (V, E')$ is, with very high probability, at most An, for some fixed $A > 0$.*

Proof: The edges of G' are edges of G that were either marked as connectors or as introvert (or both). As each contraction of a pair of vertices causes at most two edges to become connectors, and as there can be at most n such contractions, at most $2n$ edges of G are marked as connectors. We are left therefore with the task of bounding the number of edges of G marked as introvert.

Let $G_i = (V_i, E_i)$, for $0 \leq i \leq k$, be the graph at the beginning of the i-th iteration of DILUTE. Each vertex of G_i is obtained by contracting a cluster of G. Some of these clusters become introvert and are then removed. Let $h_i : V \to V_i \cup \{0\}$ be the mapping that assigns to each vertex $v \in V$ the vertex in G_i into which v was contracted, or 0 if v was contracted into a vertex that was thrown away as being introvert in one of the previous iterations.

Let $E_i^* = \{(h_i(u), h_i(v)) : (u, v) \in E \text{ and } h_i(u) \neq h_i(v) \text{ and } h_i(u), h_i(v) \neq 0\}$. The set E_i^* may contain parallel edges. Clearly $E_i \subseteq E_i^*$. Some edges of E_i^* may not appear in E_i as they have been removed during one of the sampling stages. We say that the edges of E_i are *alive* during the i-th iteration and that the edges of $E_i^* - E_i$ are *dead* during this iteration. A vertex $v \in V_i$ becomes introvert if it is adjacent to less than Δ live edges. If a vertex $v \in V_i$ becomes introvert we say that all the edges of E_i^* adjacent to v become introvert. Let F_i be the set of edges that become introvert during the i-th iteration. It is easy to see that the set $\cup_{i=0}^{k-1} F_i$ is in one to one correspondence with the set of edges marked by the algorithm as introvert after all the iterations have finished. We next obtain a bound on the number of edges that become introvert during the i-th iteration.

Lemma 7.3 *The number of edges that become introvert during the i-th iteration is, with very high probability, at most $B\alpha^i n$, for some fixed $B > 0$.*

Proof: Let $G_0, G_1, \ldots, G_k$ be the sequence of graphs obtained by running *DILUTE*. Let $V_0, V_1, \ldots$ be the sequence of vertex sets of these graphs. Let $f_0, f_1, \ldots$, where $f_i : V_i \to V_i$, be the sequence of pseudo-forests chosen, and let $M_0, M_1, \ldots$ be the sequence of matchings obtained. All these objects are in fact random variables.

We consider the *conditional* probability of having at least $B\alpha^i n$ introvert edges in the i-th iteration, *given* the sequences $V_0, V_1, \ldots, V_i$, $f_0, f_1, \ldots, f_{i-1}$ and $M_0, M_1, \ldots, M_{i-1}$. These sequences determine the sequence $E_0^*, E_1^*, \ldots, E_i^*$, but they do not determine the sequence $E_0, E_1, \ldots, E_i$ and therefore the sequence $G_0, G_1, \ldots, G_i$, as they do not determine which edges are alive and which are dead in each iteration. They do induce of course probabilities on the edges of $E_0^*, E_1^*, \ldots, E_i^*$ being either live or dead. We assume, as we can with very high probability, that $|V_i| \le \beta^i n$, for $0 \le i \le k$.

Let e be an edge of E_i^*. What is the conditional probability that e is alive at the beginning of the i-th iteration, given the sequences $V_0, V_1, \ldots, V_i$, $f_0, f_1, \ldots, f_{i-1}$ and $M_0, M_1, \ldots, M_{i-1}$? It is tempting to say that the answer is about α^i, as e has to pass i selection phases and the probability of survival in each one of them is α. This is of course an over simplification that ignores certain important factors. On the one hand, we know that e was not contracted in any of the previous iterations. This seems to decrease the probability of e being alive. On the other hand, the two endpoints of e were not declared introvert in any of the previous iterations. Each endpoint of e had, therefore, at least Δ live edges during each one of the preceding iterations. This seems to increase the probability of e being alive. A lower bound on this probability, taking all these considerations into account, is presented in the following lemma.

Lemma 7.4 *Let e be an edge of E_i^*. The probability that e is alive at the beginning of the i-th iteration, given the sequences $V_0, V_1, \ldots, V_i$, $f_0, f_1, \ldots, f_{i-1}$ and $M_0, M_1, \ldots, M_{i-1}$ is at least $\frac{1}{2}\alpha^{3i}$.*

Proof: Let Γ be the event that corresponds to the sequences $V_0, V_1, \ldots, V_i$, $f_0, f_1, \ldots, f_{i-1}$ and $M_0, M_1, \ldots, M_{i-1}$. Let W_0 denote the event that e is dead at the beginning of the i-th iteration, i.e., that $e \notin E_i$, and let W_1 denote the event that e is alive at the beginning of the i-th iteration, i.e., that $e \in E_i$. We have to show that $\Pr[W_1 \mid \Gamma] \ge \frac{1}{2}\alpha^{3i}$.

A *point* ω of the sample space is an atomic event in which the outcomes of all the random decisions of the algorithm are specified. These random decisions include the decisions made during the sampling stage, the selection of the pseudo-forests, and the random decisions made by the random mating stage of *COUPLE*.

Let W_0^j, for $0 \le j < i$, be the event that the edge e was killed during the j-th iteration. Clearly $W_0 = \cup_{j=0}^{i-1} W_0^j$. Let $\omega_0^j \in W_0^j \cap \Gamma$. We construct a point $\omega_1 \in W_1 \cap \Gamma$ in the following way. The point ω_1 is identical to ω_0^j, except that e survives the sampling stages in iterations $j, \ldots, i-1$ and dies in the sampling stage of iteration i. Note that e is alive in ω_1 in the beginning of the i-th iteration. It is easy to check that ω_1 corresponds to a possible run of *DILUTE* and it is therefore a point in the sample space and in the event $W_1 \cap \Gamma$. We claim that

$$\Pr[\omega_1] \ge \alpha^{i-j} \cdot \left(\frac{\Delta}{\Delta+1}\right)^{2(i-j)} \cdot \Pr[\omega_0^j] \,.$$

The α^{i-j} accounts for the fact e is now to survive $i-j$ further sampling stages before dying. The pseudo-forests chosen in ω_1 should be the same as those chosen in ω_0^j. Let u_l and v_l be the endpoints of e at the beginning of the l-th iteration, for some $j \le l < i$. Let $\deg(u_l)$ and $\deg(v_l)$ be the degrees of u_l and v_l, according to ω_0^j, in the beginning of the l-th iteration. We know that $\deg(u_l), \deg(v_l) \ge \Delta$. Let e_{u_l} and e_{v_l} be the edges chosen for the pseudo-forest by u_l and v_l in the l-th iteration. The probabilities of these choices, in ω_0^j, are $1/\deg(u_l)$ and $1/\deg(v_l)$, as the edges are chosen to be in the pseudo-forest with equal probabilities. In ω_1, the edge e is alive and the degrees of u_l and v_l are now $\deg(u_l)+1$ and $\deg(v_l)+1$. The probabilities that e_u and e_v will be chosen now are only $1/(\deg(u_l)+1)$ and $1/(\deg(v_l)+1)$, respectively. The bound follows from the fact that

$\deg(u_l)/(\deg(u_l)+1), \deg(v_l)/(\deg(v_l)+1) \geq \Delta/(\Delta+1)$ and the fact that $2(i-j)$ identical decisions should be made. Using the fact that $\Delta \geq \alpha/(1-\alpha)$ we get that $\Delta/(\Delta+1) \geq \alpha$ and therefore

$$\Pr[\omega_1] \geq \alpha^{3(i-j)} \cdot \Pr[\omega_0^j] .$$

The transformation F from $\omega_0^j \to \omega_1$ is a one to one transformation from $W_0^j \cap \Gamma$ to $W_1 \cap \Gamma$. It is not one to one however from $W_0 \cap \Gamma$ to $W_1 \cap \Gamma$. For a point $\omega_1 \in W_1$ there may exist a sequence of points $\omega_0^0, \omega_0^1, \ldots, \omega_0^{i-1}$, such that $\omega_0^j \in W_0^j$ and $F(\omega_0^j) = \omega_1$. For some j's, the point ω_0^j may not exist. If a point ω_0^j does not exist, we let $\Pr[\omega_0^j] = 0$. An easy manipulation then gives us

$$\Pr[\omega_1] \cdot \frac{\alpha^{-3i}-1}{1-\alpha^3} \geq \Pr[\omega_1] \cdot \sum_{j=0}^{i-1} \alpha^{3(j-i)} \geq \sum_{j=0}^{i-1} \Pr[\omega_0^j] .$$

As a consequence we get that $\Pr[W_1 \cap \Gamma] \geq \frac{1-\alpha^3}{\alpha^{-3i}-1} \cdot \Pr[W_0 \cap \Gamma]$ and therefore that

$$\Pr[W_1 \mid \Gamma] \geq \frac{1-\alpha^3}{\alpha^{-3i}-1} \cdot \Pr[W_0 \mid \Gamma] .$$

As $\Pr[W_0 \mid \Gamma] + \Pr[W_1 \mid \Gamma] = 1$, we get that

$$\Pr[W_1 \mid \Gamma] \geq \frac{1-\alpha^3}{\alpha^{-3i}-\alpha^3} \geq (1-\alpha^3)\alpha^{3i} > \frac{1}{2}\alpha^{3i} ,$$

as required. □

Considering again the proof just given, we see that if $e_1', \ldots, e_l'$ are edges of E_i^* distinct from e, and if $e_1', \ldots, e_l'$ are dead in the beginning of the i-th iteration according to ω_0^j, then these edges are also dead in the start of the i-th iteration according to ω_1. We therefore obtain the following strengthened version of Lemma 7.4.

Lemma 7.5 *Let $e_1, \ldots, e_l$ be distinct edges of E_i^*. The probability that e_1 is alive at the beginning of the i-th iteration, given the sequences $V_0, V_1, \ldots, V_i$, $f_0, f_1, \ldots, f_{i-1}$ and $M_0, M_1, \ldots, M_{i-1}$ and given that $e_2, \ldots, e_l$ are dead in the beginning of the i-th iteration, is at least $\frac{1}{2}\alpha^{3i}$.*

We now return to the proof of Lemma 7.3. We prove that for any sequences $V_0, V_1, \ldots, V_i$, $f_0, f_1, \ldots, f_{i-1}$ and $M_0, M_1, \ldots, M_{i-1}$, the conditional probability that more than $B\alpha^i n$ edges become introvert during the i-th iteration is exponentially small. The claim of the Lemma follows immediately from this claim.

Consider the vertex set V_i of the graph G_i at the start of the i-th iteration. Some of the vertices of V_i become introvert vertices during the i-th iteration and all the edges adjacent to them become introvert edges. Let us bound the probability that a specific subset $U \subseteq V_i$ becomes introvert and that as a result at least $y_i = B\alpha^i n$ edges become introvert. Multiplying this probability by 2^{n_i}, the number of possible subsets of vertices that may become introvert, me obtain a bound on the probability of obtaining at least $y_i = B\alpha^i n$ introvert edges in the i-th iteration.

Let $U = \{v_1, \ldots, v_l\}$ be a subset of V_i. Let F_i be the set of edges adjacent to at least one vertex from U. The set F_i is the set of edges that become introvert if the vertices of U, and only them, become introvert. We assume that $|F_i| \geq y_i$. Let $F_i' \subseteq F_i$ be a fixed subset of F_i of size exactly y_i. Let m_j be the number of edges from F_i' that are adjacent to v_j. If the vertices of U are to become introvert then all the edges of F_i', except for at most Δ edges touching each vertex of U must be dead after the sampling stage of the i-th iteration. Let F_i'' be a subset of F_i' obtained by removing at most Δ edges that are adjacent to each vertex of U. The size of F_i'' is at least $y_i - \Delta l$ and there are at most $\prod_{j=1}^{l}(\sum_{\nu=0}^{\Delta}\binom{m_j}{\nu}) \leq \prod_{j=1}^{l}(2m_j)^{\Delta}$ choices for F_i'' (we may assume that $m_j > 0$ for every $1 \leq j \leq l$). Using the arithmetic-geometric mean inequality and using the fact that $\sum_j m_j \leq 2y_i$, we get that the number of these choices is at most $(4y_i/l)^{\Delta l}$. Using Lemma 7.5 we get that the probability that all the edges of F_i'' are dead after the sampling stage of the i-th iteration is at most $(1-\frac{1}{2}\alpha^{3i+1})^{y_i-\Delta l} \le e^{-\frac{1}{2}(y_i-\Delta l)\alpha^{3i+1}}$.

Let I_i be the random variable giving the number of edges that become introvert during the i-th iteration. Let Γ be as in the proof of Lemma 7.4. We get therefore that

$$\Pr[I_i \ge y_i \mid \Gamma] \le 2^{n_i} \cdot \left(\frac{4y_i}{l}\right)^{\Delta l} \cdot e^{-\frac{1}{2}(y_i-\Delta l)\alpha^{3i+1}} .$$

The function $(c/x)^x$ is increasing for $x \le c/e$. As $l \le n_i \le 4y_i/e$, we get that $(4y_i/l)^{\Delta l} \le (4y_i/n_i)^{\Delta n_i}$ and therefore

$$\Pr[I_i \ge y_i \mid \Gamma] \le 2^{n_i} \cdot \left(\frac{4y_i}{n_i}\right)^{\Delta n_i} \cdot e^{-\frac{1}{2}(y_i-\Delta n_i)\alpha^{3i+1}}$$

Taking the logarithm of this expression, and recalling that $n_i \le \beta^i n$ and that $y_i = B\alpha^i n$ we get

$$[\ln 2\cdot\beta^i + (\ln 4B + i\ln\frac{\alpha}{\beta})\cdot\Delta\beta^i + \tfrac{1}{2}\Delta\beta^i\alpha^{3i+1} - \tfrac{1}{2}B\alpha^{4i+1}]\cdot n$$

As $\alpha^4 > \beta$, the dominant term in this expression is $-\frac{1}{2}B\alpha^{4i+1}$. If B is chosen to be large enough, the whole expression is negative for all values of $i \ge 0$. It can be easily checked, for example, that if $B = 2000$ then the above expression is always smaller than $-40\alpha^{4i}n$. With this choice of parameters we get therefore that $\Pr[I_i \ge y_i \mid \Gamma] \le e^{-40\alpha^{4i}n}$. Note that as $i \le k$ and $\alpha^4 > \beta$, we get that $\alpha^{4i}n \ge n/\log^4 n_0$. This completes the proof of Lemma 7.3. □

We finally return to the proof of Theorem 7.2. The number of edges marked as connectors is at most $2n$. The number of edges marked as introvert is at most $\sum_{i=0}^{k-1} B\alpha^i n \le \frac{B}{1-\alpha}n$. The Theorem then follows by taking $A = \frac{B}{1-\alpha} + 2 = 200002$. □

7.1 The implementation of *DILUTE*

The implementation of *DILUTE* is relatively straightforward. We assume again that a virtual processor is initially allocated to edge and non-isolated vertex of G_0, the input graph to *DILUTE*. When an edge is deleted and when a vertex becomes isolated, the processors associated with them become idle. This implementation that ignores the processors allocation issue is then turned into a concrete implementation using the load balancing scheme of Section 4.

Let n_i be the number of vertices in G_i, let m_i be the number of edges of G_i and let d_i be the maximal degree of G_i. It is easy to see that $n_i \le \beta^i n$ and that $d_i \le d2^i$, where n is the number of vertices of G_0 and d is the maximal degree of G_0. Because of the sampling stage at the beginning of each iteration, we get that with very high probability $m_i \le 2\alpha^i m$, where m is the number of edges in G_0.

The edge sampling and the marking and throwing away of the introvert vertices in the i-th iteration are easily implemented in $O(\log d_i)$ time and $O(m_i \log d_i)$ work. The call to *COUPLE* in the i-th iteration requires $O(\log d_i)$ time and $O(m_i \log d_i)$ work. The total time required by the `for` loop of *DILUTE* is therefore

$$O(\sum_{i=0}^{k-1} \log d_i) = O(\sum_{i=0}^{k-1}(\log d + i)) = O((\log d)\cdot\log\log n + (\log\log n)^2)$$

and the total work is

$$O(\sum_{i=0}^{k-1} m_i \log d_i) = O(\sum_{i=0}^{k-1} \alpha^i m\cdot(\log d + i)) = O(\log d\cdot\sum_{i=0}^{k-1}\alpha^i m + \sum_{i=0}^{k-1} i\alpha^i m) = O(m\log d) .$$

The calls to *REDUCE* during the `for` loop of *DILUTE* produce an array h. If vertex v is contracted in one of the iterations into vertex u, then $h(v) = u$. A vertex v is contracted in at most one of the iterations. In each one of the iterations at most one vertex is contracted into v. The array h defines a forest. The depth of this forest is at most $k = O(\log\log n)$. We are supposed to compute an array h^* so that for every vertex v, $h^*(v)$ is the root of the tree containing v. The array h^* is constructed in the following way. For every vertex $v \in V_k$ we let $h^*(v) \leftarrow v$. We then perform a second loop in which i ranges from $k-1$ *downto* 0. If u was contracted into v in the i-th iteration of the first loop, then in the i-th iteration of the second loop, the *actual* processor allocated to u during the i-th iteration of the first loop performs $h^*(u) \leftarrow h^*(v)$. It is easy to check that this produces the array h^* as required.

All the subsequent operations of the procedure *DILUTE* are then performed in $O(\log d)$ time using $O(m \log d)$ work. The total running time of *DILUTE* is therefore $O((\log d)\cdot\log\log n + (\log\log n)^2)$ and the total work is $O(m\log d)$, as promised.

7.2 A comparison with Gazit's algorithm

The procedure *DILUTE* forms the heart of the first stage of our algorithm. The general structure of *DILUTE* is similar to the *partitioning* procedure used by Gazit [Gaz91]. Gazit's procedure however runs on the much stronger CRCW PRAM. Many intricate changes had to be made to obtain a version of this procedure that could run on an EREW PRAM.

In the (Arbitrary) CRCW PRAM model used by Gazit, the degrees of the vertices of the graph have no influence on the running time of the algorithm. The situation is completely different in the EREW PRAM model. To avoid concurrent reads and writes, all the operations performed on a graph should use its adjacency lists. The time required to perform an operation is usually determined by the maximal degree of the graph. To obtain a logarithmic running time the degrees of the vertices have to be carefully controlled.

To solve the processor allocation problem of his algorithm, Gazit uses a CRCW PRAM prefix sums algorithm of Cole and Vishkin [CV89] that runs in $O(\frac{\log n}{\log\log n})$ time. On an EREW PRAM, the running time of any prefix sums algorithm is $\Omega(\log n)$ ([CDR86],[DKR94]). Again, an alternative had to be found. Our algorithm uses instead the load balancing scheme described in Section 4.

The contractions in Gazit's algorithm are performed using a *deterministic mating* procedure. This procedure finds stars and contracts them. The degrees of these stars are unbounded so we could not adopt the same approach. As a replacement, we use the procedure *COUPLE* that finds pairs of vertices instead of stars.

The sampling stage in our algorithm is also different from the sampling stage in Gazit's algorithm. At the i-th iteration Gazit chooses a random sample of $\alpha^i m$ edges. The sample is chosen from the *original* edge set of the graph. Edges that were not sampled in some iteration may be sampled in a subsequent iteration. The sample we choose in the i-th iteration is always a subset of the sample chosen in the $i-1$-th iteration. We cannot adopt Gazit's approach as constructing the adjacency lists of the sampled graph will involve too much work.

The fact that our samples are not independent extremely complicates the correctness proof of our algorithm (cf. Lemma 7.3). To make the proof go though we have to change the definition of introvert vertices. In Gazit's algorithm an introvert vertex is a vertex that has no sampled edges adjacent to it. In our algorithm an introvert vertex is a vertex with less than $\Delta = 100$ live (i.e., sampled) edges adjacent to it. Our proof makes an essential use of the fact that the construction of the pseudo-forest used to obtain the matching is randomised and that each edge has a small enough probability (at most $1/\Delta$) of being chosen. Our proof uses the fact that $\Delta \geq \alpha/(1-\alpha)$ and it does not work therefore for $\Delta = 1$, as in Gazit's algorithm. It is possible however that our algorithm remains correct even when $\Delta = 1$ but a different proof is needed in order to show it.

8 The procedure *IMAGE*

The input to *IMAGE* is composed of a graph $G = (V, E)$ and a partial image $G' = (V', E')$ of it along with the image function $h : V \to V'$ and the inverse image function $h^{-1} : V' \to 2^V$. The image function h gives for every vertex $v \in V$ the vertex $h(v) \in V'$ to which v is mapped. The inverse image function h^{-1} gives for every vertex $v' \in V'$ a list containing the vertices of V mapped into v'. The procedure *IMAGE* returns the graph $G'' = (V', E'')$ where $E'' = \{(h(u), h(v)) : (u, v) \in E \text{ and } h(u) \neq h(v)\}$.

Each vertex $v \in V$ has a list of the edges adjacent to v. Each edge (v, u) has a pointer to its opposite edge (u, v). Each vertex $v' \in V'$ has a linked list containing the vertices of V mapped into v', i.e., the vertices of the set $h^{-1}(v')$.

The graph G'' is constructed as follows. Each vertex v distributes its image $h(v)$ to all the edges adjacent to it. Each edge learns in this way the image under h of its first endpoint. Each edge then learns the image under h of its second endpoint by sharing information with its opposite edge. Each edge (u, v) is thus replaced by an edge $(h(u), h(v))$. Edges for which $h(u) = h(v)$ are marked for deletion and then deleted by compressing the adjacency lists of G.

To generate the adjacency lists of G'' we have to concatenate the adjacency lists of all the vertices of G that are mapped into the same vertex of V'. This is easily done using the linked list $h^{-1}(v')$ that we have for each vertex $v' \in V'$. Note that we do *not* remove parallel edges from G'' as this requires more than linear work. The graph G'' may therefore be a multigraph but this causes no problems in *SPARSIFY*.

All these operations can be performed without any concurrent reads and writes in $O(\log n)$ time using $O((m+n)/\log n)$ processors, where $n = |V|$ and $m = |E|$, using optimal list ranking and prefix sums algorithms.

9 The procedure *SPARSIFY*

The procedure *SPARSIFY* is based on the sampling Lemma of Karger, Klein and Tarjan [KKT95]. This Lemma is the key to their randomised linear-time algorithm for finding minimum spanning trees. The original version of this Lemma was obtained by Karger [Kar93],[KNP92]. Klein and Tarjan [KT94] then obtained the tighter version that can also be found in [KKT95].

As our graph is unweighted, we only need the following simplified version of the sampling Lemma of Karger, Klein and Tarjan. For completeness, we include a proof of this version. Throughout this section, all graphs may contain parallel edges.

Lemma 9.1 *Let $G = (V, E)$ be a graph with n vertices and m edges and let $0 < p < 1$. Let $G' = (V, E')$ be a random subgraph of G obtained by including each edge of E in E' independently with probability p. Then, the probability that the number of edges of E that connect vertices in different connected components of the subgraph $G' = (V, E')$ exceeds $2n/p$ is at most $e^{-n/4}$.*

Proof: Let $e_1, e_2, \ldots, e_m$ be the edges of E in some order. An edge is said to be *selected* if it is in E'. Let E'_i be the set of selected edges among $e_1, e_2, \ldots, e_i$. An edge e_i is said to be a *bridge* if it connects different connected components of the graph (V, E'_{i-1}). The number of bridges obtained is clearly an upper bound on the number of edges in E that connect different connected components of $G' = (V, E')$. We therefore bound the probability of obtaining at least $k = 2n/p$ bridges.

It is easy to see that the number of selected bridges can be at most $n-1$ as afterwards the subgraph composed of the selected edges is completely connected. For every string $w = w_1w_2 \ldots w_k \in \{0, 1\}^k$, consider the event A_w in which the i-th bridge, for $1 \leq i \leq k$, is selected if and only if $w_i = 1$. Some of these events, like those in which w contains more than $n-1$ ones, are empty. Note that the decisions whether or not to include each one of the first i bridges determine uniquely whether there will be an $i+1$-st bridge and if so which edge in the sequence this would be. Thus, each event A_w is either empty, or there is a sequence $b_1 < b_2 < \ldots < b_k$ of indices such that the event A_w occurs if and only if the edge e_{b_i} is selected whenever $w_i = 1$ and unselected otherwise. The union of all the events A_w, for every $w \in \{0,1\}^k$, is the event of having at least k bridges.

Suppose that $w \in \{0,1\}^k$ and that w contains j ones and $k-j$ zeros. If the event A_w is not empty then $j \leq n-1$. As the decision whether to select the i-th bridge is independent of all the previous selections, the probability of the event A_w is at most $p^j(1-p)^{k-j}$. The probability of obtaining at least k bridges is therefore bounded by $\sum_{j=0}^{n-1}\binom{k}{j}p^j(1-p)^{k-j}$. This is exactly the probability that a binomial random variable with mean $pk = 2n$ gets a value below n. Using a standard Chernoff bound (see Alon and Spenser [AS92], p. 237) it is easy to see that this probability is bounded by $e^{-n/4}$. □

The procedure *SPARSIFY* receives a graph $G = (V, E)$ with n vertices and m edges. It builds a random subgraph $G' = (V', E')$ of G by including each edge of E in E' independently with probability $p = 1/\log^2 n$ and then removing the isolated vertices. The graph G' will have, with very high probability, at most $O(m/\log^2 n)$ vertices and edges. We then find the connected components of G' using the procedure *CONNECT* that constitutes the second stage of our algorithm. This takes $O(\log n)$ time using $O((n+m)/\log n)$ processors. The procedure *CONNECT* outputs an image $G'' = (V'', \phi)$ of G' together with an image function $h' : V' \to V''$ and the inverse image function $(h')^{-1} : V'' \to 2^{V'}$. A call to *IMAGE*$(G, G'')$ produces an image $G'' = (V, E'')$ of G'. It follows from Lemma 9.1 that the number of edges in G'' is, with very high probability, at most $O(n \log^2 n)$, as required.

PART II : FINDING THE CONNECTED COMPONENTS

10 An overview of the second stage

In this part of the paper we describe the second stage of our algorithm. The input to this stage is a graph $G = (V, E)$ with n vertices and m edges. We are supposed to find the connected components of this graph in $O(\log n)$ time using $O((n+m)\log n)$ processors. We achieve this goal using an adaptation of the algorithm of Karger *et al.* [KNP92]. Before describing our algorithm we therefore review, in the next subsection, the algorithm of Karger *et al.* [KNP92]. An overview of our algorithm is then given in Subsection 10.2. The detailed description of our second stage is then given in Sections 12 to 16.

10.1 The algorithm of Karger, Nisan and Parnas

Most parallel connectivity algorithms find the connected components of the input graph by performing a sequence of *hooking* and *contraction* phases. In a hooking phase each vertex v sets a pointer $f(v)$ to one of the neighbours of v in the graph. It is usually required that the graph defined by these pointers will be a forest. In a contraction phase, each tree of the forest obtained in the preceding hooking phase is contracted into a single vertex.

Let $G = (V, E)$ be a graph on n vertices numbered $1, 2, \ldots, n$. Assume that the degree of each vertex of G is at least d. Let $N(v)$ denote the set of neighbours in G of a vertex $v \in V$. We consider a vertex v to be a neighbour of itself and therefore $v \in N(v)$. Consider the following hooking scheme. Each vertex $v \in V$ finds the largest vertex at a distance of at most two from itself, i.e., the largest vertex in $N(N(v)) = \cup_{u \in N(v)} N(u)$, and hooks to it. This can be implemented in two rounds. In each round each vertex considers the vertices adjacent to it. A vertex hooks to itself if there is no vertex larger than it in $N(N(v))$. As each vertex hooks either to itself or to a vertex with a larger serial number, no cycles are formed and the pointers chosen define a forest. It is easy to check that the distance between two vertices that are roots in this forest is greater than two. The neighbours of the roots are thus disjoint and the number of trees in the forest is at most n/d.

In graphs with a large minimal degree, each such hooking iteration greatly reduces the number of vertices in the graph and the algorithm will complete after a relatively small number of iterations. The algorithm may require many iterations however, if the input graph has many vertices of small degree.

The above hooking scheme will work correctly if the neighbourhood $N(v)$ of each vertex is replaced by a *virtual neighbourhood* $C(v)$, provided that all the vertices of $C(v)$ are in the connected component of v and that the virtual neighbourhoods defined are symmetric, in the sense that $u \in C(v)$ if and only if $v \in C(u)$. We are thus left with the task of finding for each vertex v of the graph a large virtual neighbourhood.

Karger, Nisan and Parnas [KNP92] construct the virtual neighbourhoods using relatively short random walks. A similar approach was used by Aleliunas *et al.* [AKL$^+$79] and recently by Nisan, Szemeredi and Wigderson [NSW92] and Barnes and Feige [BF93] to obtain space efficient algorithms and time-space tradeoffs for undirected s-t connectivity.

Barnes and Feige [BF93], proving a conjecture of Linial, showed that a random walk of length s in an undirected connected graph is likely to visit at least $\Omega(s^{1/3})$ vertices (or all the vertices of the graph). A slightly weeker bound of $\Omega(s^{1/4})$ appears in Karger *et al.* [KNP92].

The algorithm of Karger *et al.* [KNP92] initiates random walks of length $\Theta(n^\epsilon)$, for some $\epsilon > 0$, from each of the n vertices of the graph G. A vertex u belongs to the virtual neighbourhood $C(v)$ of vertex v if the random walk that starts at v passes through u or if the random walk that starts at u passes through v. The size of each virtual neighbourhood $C(v)$ is, with very high probability, at least $\Omega(n^{\epsilon/3})$, or otherwise, $C(v)$ contains all the vertices in the connected component of v, which is even better. The algorithm would therefore terminate after $O(1/\epsilon)$ hooking and contraction phases. The description above is slightly over simplified. Due to implementation reasons, the exact definition of the virtual neighbourhoods is slightly more complicated.

Karger *et al.* implement each hooking phase, including the random walks, in $O(\log n)$ time using $O(n^{1+\epsilon} + m)$ processors. Each contraction phase is easily implemented in $O(\log n)$ time using $O((n+m)/\log n)$ processors. As their algorithm uses only a constant number of phases, their whole algorithm runs in $O(\log n)$ time using $O(n^{1+\epsilon} + m)$ processors. Karger *et al.* also describe a slightly improved version of their algorithm that uses only $O((n^{1+\epsilon} + m)/\log n)$ processors.

10.2 Our connected components algorithm

The algorithm of Karger *et al.* uses $O(1)$ phases of random walks. The random walks used in each phase are all of length $\Theta(n^\epsilon)$. The implementation of each phase of random walks takes $O(\log n)$ time using $O(n^{1+\epsilon} + m)$ processors.

Our algorithm performs about $O(\log\log n)$ phases of random walks. The first random walks performed are extremely short and require substantially less than $O(\log n)$ time. The random walks become longer and longer as the number of non-isolated vertices in the graph becomes smaller and the number of processors allocated to each such vertex becomes larger. The lengths of the random walks performed form a doubly exponential sequence. We use a simple scheme to allocate more and more processors to the surviving vertices of the graph.

Our second stage begins by obtaining an image of the input graph in which the maximal degree is at most three. Again, we carefully control the growth of the degrees in the graph. The first random walk phases are therefore performed on graphs with small maximal degrees. A phase of random walks of length s starting from each vertex of a graph containing n vertices, m edges and maximal degree d is implemented in $O(\log s + \log d)$ time using $sn + m$ processors.

The input to the second stage is a graph $G = (V, E)$ with n vertices and m edges. We are to find its connected components in $O(\log n)$ time using $O((m + n)\log n)$ processors. The graph G is immediately transformed, as already mentioned, into a graph with $O(n + m)$ vertices and edges with maximal degree at most three. We assume therefore, for simplicity, that the input graph $G = (V, E)$ is already of this form, i.e., that the maximal degree of G is at most three and that therefore $m \leq 1.5n$. We are then to find the connected components of G in $O(\log n)$ time using $O(n \log n)$ processors.

The simple data structures that allow us to allocate more and more processors to the surviving vertices of the graph are described in the next subsection. A high level description of the procedure *CONNECT* that constitutes the second stage of our algorithm is then given in Section 11.

10.3 Graph organisation

As in the first stage of our algorithm, the graph $G = (V, E)$ is represented using its adjacency lists. Each vertex v has an array $N[v]$ in which the edges adjacent to v are stored in consecutive memory locations. Each edge (v, u) in $N[v]$ has a pointer to its opposite edge (u, v) in $N[u]$. The graph G is changed during the course of the algorithm. We always use G to refer to the current state of the graph and not to its original state.

We have $\Theta(n \log n)$ processors at our disposal. We allocate $\Theta(\log n)$ processors to each vertex of G and a single processor to each edge of G. Each processor allocated to a vertex knows its serial number among the processors allocated to that vertex.

The vertices of G are organised in stars. The leaves of these stars get the attribute DONE. The roots of these stars get the attribute STAR. Each DONE vertex v has a pointer $\mathit{root}(v)$ to the root of its star. If u is a STAR vertex then $\mathit{root}(u) = u$. Each STAR vertex u has an array $\mathit{vertices}[u]$ that holds a list of all the vertices in its star (including itself) and a field $\mathit{size}(u)$ that holds the length of this list (i.e., the size of the star). Each vertex v in $\mathit{vertices}[u]$ knows its serial number in the list. In this way a STAR vertex v can be easily assigned with $\mathit{size}(v) \cdot \log n$ processors. At the beginning of the algorithm all vertices get the attribute STAR.

When a star is formed, by the amalgamation of some existing stars, the adjacency lists of all the vertices participating in it are amalgamated into a single adjacency list. An edge (x, y) of the original graph is converted into the edge $(\mathit{root}(x), \mathit{root}(y))$. A STAR vertex v has an array $N[v]$ that holds the list of all the edges that connect the STAR v to the other STARs. There is a pointer from the edge (u, v) in $N[u]$ to the edge (v, u) in $N[v]$ and vice versa. The degree of v is the length of the list $N[v]$. As the original graph has maximal degree 3, the degree of a STAR v is at most $3 \cdot \mathit{size}(v)$. When the adjacency list of a star is formed, internal and parallel edges are removed. The processors allocated to removed edges become inactive.

The STAR vertices of the graph are organised in trees. Each STAR vertex v has a pointer $\mathit{parent}(v)$ to another STAR vertex of the graph. If v is a root of a tree in the forest thus formed then $\mathit{parent}(v) = v$. Each STAR vertex v has an array $\mathit{children}[v]$ that holds a list of all the STAR vertices that point to it. Each STAR vertex u in the list $\mathit{children}[v]$ has a pointer to its place in the list. The size $\mathit{size}(T)$ of a tree T is defined as the sum of the sizes of all the STAR vertices in that tree. Clearly, the super-vertices in a tree T have together at most $3 \cdot \mathit{size}(T)$ neighbours. At the beginning of the algorithm each STAR vertex v forms a tree of its own and $\mathit{parent}(v) = v$.

11 The procedure *CONNECT*

The procedure *CONNECT* that finds the connected components is described in Figure 4. The procedure *CONNECT* calls the procedures *CONTRACT* and *HOOK* described in Sections 12 and 13.

```
CONNECT(G)
input: A graph G = (V, E) with n vertices, m ≤ 1.5n edges and maximal
  degree at most three.
output: The connected components of G.
complexity: O(log n) time using O(n log n) processors.
s_0 ← 2
for i ← 0 to k − 1 (where k = O(log log n)) do
begin
{
    s_{i+1} ← s_i^{4/3}
    CONTRACT(s_{i+1})
    HOOK(s_i)
}
end
```

Figure 4: The procedure *CONNECT*.

To get the algorithm started we perform a single step, not shown in Figure 4, in which every vertex hooks on the largest vertex at a distance at most two from itself. This can be easily implemented in $O(1)$ time as the degree of each vertex is at most three. Every non-isolated vertex in G is now contained in a tree whose size is at least two.

Define the sequence $s_0 = 2$ and $s_{i+1} = s_i^{4/3}$. Clearly, $s_i = 2^{(4/3)^i}$. The algorithm performs $k = \lceil \log_{4/3} \log n \rceil = O(\log \log n)$ iterations. Note that $s_k \geq n$ and that $(4/3)^k = O(\log n)$. The i-th iteration includes a contraction phase in which trees whose sizes are at most s_{i+1} are contracted, followed by a hooking phase that uses random walks of length $\Theta(s_i)$.

Before the i-th iteration of the algorithm, where $1 \leq i \leq k$, is executed, the size of each tree in G, that does not correspond to a complete connected component of G, is at least s_i. The size of each STAR vertex, however, is at most s_i.

The i-th iteration starts by a call to $\mathit{CONTRACT}(s_{i+1})$. This call contracts each tree in G whose size is at most s_{i+1} into a single STAR vertex. The STAR vertices that correspond to trees that were contracted during this iteration get the attribute ACT (active). All STAR vertices contained in trees that were not contracted get the attribute INA (inactive). ACT vertices that correspond to complete connected components are removed from the graph. The size of each remaining ACT vertex is at least s_i, as we assumed that the size of each tree in G (that does not correspond to a complete connected component of G) at the beginning of the i-th iteration is at least s_i. The execution of these operations takes $O(\log s_{i+1} + \log \log n)$ time. This part of the algorithm is reminiscent of the algorithms of Johnson and Metaxas [JM91],[JM92] and Chong and Lam [CL95]

A call is next made to $\mathit{HOOK}(s_i)$. This call starts $\Theta(\log n)$ independent random walks of length $\Theta(s_i)$ from each ACT vertex of G. Each such random walk is stopped if it reaches an INA vertex. As the size of each ACT vertex is at least s_i, each ACT vertex has at least $s_i \cdot \log n$ processors allocated to it. As the size of each ACT vertex is at most s_{i+1}, its degree is at most $3 \cdot s_{i+1}$. These random walks can therefore be implemented in $O(\log s_{i+1} + \log \log n)$ time.

Using these random walks we obtain, with very high probability, for every ACT vertex, a virtual neighbourhood that is either of size at least $s_i^{1/3}$, or contains an INA vertex, or contains all the vertices of the corresponding connected component.

Using these virtual neighbourhoods, we hook ACT vertices on either INA vertices or other ACT vertices so that the size of each tree that does not correspond to a complete connected component is, with very high probability, at least s_{i+1}. Our hooking method is slightly different from that used by Karger *et al.* [KNP92] as we want to ensure not only that the number of trees formed is small enough but also to ensure that each tree formed is large enough. The hooking process takes again $O(\log s_{i+1} + \log\log n)$ time. The $O(\log\log n)$ term accounts for the fact that we have to consider the results of $\Theta(\log n)$ independent random walks from each vertex. When the hooking process is done we ignore the ACT and INA attributes. This completes the i-th iteration. The graph now satisfies the conditions that should be met before the $i+1$-st iteration.

The i-th iteration takes $O(\log s_{i+1} + \log\log n)$ time, using $O(n \log n)$ processors. The total running time of the second stage is therefore

$$O(\sum_{i=0}^{k}(\log s_{i+1} + \log\log n)) = O(\sum_{i=1}^{k}(4/3)^i) + O((\log\log n)^2) = O(\log n)\ .$$

After these $k = O(\log\log n)$ iterations are performed we are left with a collection of stars. Each star corresponds to a complete connected component of the original graph.

We now describe in some more detail the contractions, the random walks and the hookings phases.

12 The procedure *CONTRACT*

The input to the contraction phase is a graph in which each tree of STARs is of size at least s_i (ignoring trees that correspond to complete connected components) and each STAR is of size at most s_i. The degree of each STAR is therefore at most $d = 3 \cdot s_i$. We are supposed to contract each tree T whose size is at most $s = s_{i+1}$ and replace it by a single STAR vertex. If r is the root of a tree T of size at most s, then r will stay a STAR vertex. All the other STAR vertices of T, including the DONE vertices of these stars will now point to r. The adjacency arrays of all the STAR vertices in T will be amalgamated into a single adjacency array $N[r]$. During this amalgamation, duplicate and parallel edges will be removed. The arrays *vertices*$[v]$ of the STAR vertices v of T will also be amalgamated into the single array *vertices*$[r]$. The array *children*$[r]$ will be reset to contain only r itself and *parent*(r) will be set to r. This is again reminiscent of the algorithm of Chong and Lam [CL95].

We shortly sketch the implementation details of the contraction phase. Using the *children* arrays of the STAR vertices we can easily define an *Euler tour* in each tree. This takes only $O(1)$ time. We then perform the first $O(\log s)$ steps of the naive *list ranking* algorithm in an attempt to sum the sizes of the STAR vertices in each tree. Each STAR vertex can then know whether it belongs to a tree of size at most s (note that a tree of size at most s contains in particular at most s STAR vertices). We then use the list ranking algorithm to compute the *prefix sums* of the sizes of STAR vertices contained in each small enough tree. This again takes only $O(\log s)$ time. It is then easy to amalgamate the lists *vertices*$[v]$ associated with the STAR vertices in each tree. The amalgamation takes $O(\log d)$ time as $O(\log d)$ time is needed in the EREW PRAM model to distribute the corresponding prefix sum to the at most $s_i \le d$ DONE vertices associated with each STAR vertex. In a similar manner we amalgamate the arrays $N[v]$ associated with the STAR vertices in each tree. This amalgamation again takes $O(\log s + \log d)$ time. If v is a STAR vertex in a tree T that was contracted to its root r, then every edge (u, v) connected to v should now be replaced by an edge (u, r). This is done using the pointer from (v, u) in $N[v]$ to (u, v) in $N[u]$. Internal edges, i.e., edges between two STAR vertices contained in a contracted tree, are removed. All the adjacency lists are now sorted and parallel edges are removed. As the length of each adjacency list is at most $3s$, the sorting takes only $O(\log s)$ time. The total time required by the contraction stage is therefore $O(\log s + \log d) = O(\log s_{i+1})$. For all these operations we use only one processor for each vertex of the graph and at most one processor for each edge.

13 The procedure *HOOK*

The input to the hooking phase is a graph produced by the contraction phase. The forest defined by the *parent* pointers is composed of trees whose size is at least s_{i+1} (ignoring again trees that correspond to complete connected components) and STAR vertices that result from trees contracted during the contraction phase. Each such STAR vertex is of size at least s_i and at most s_{i+1}. These STAR vertices have the attribute ACT. The STAR vertices contained in trees all have the attribute INA. Each ACT vertex v has at least $size(v) \cdot \log n \geq s_i \cdot \log n$ processors allocated to it and its degree is at most $3s_{i+1}$.

Using $\Theta(\log n)$ independent random walks of length $\Theta(s_i)$, we obtain for every ACT vertex v a virtual neighbourhood that, with very high probability, either includes at least $s_i^{1/3}$ vertices, or includes an INA vertex, or includes all the vertices in the connected component of v. We also make sure that the virtual neighbourhood of v will not be *too big*, so that the vertex with the largest serial number in this neighbourhood could be found in $O(\log s_{i+1} + \log\log n)$ time. The way these neighbourhoods are obtained is described in the next section. Their construction takes again $O(\log s_{i+1} + \log\log n)$ time.

Each STAR vertex in G has an associated serial number. We want the serial numbers of the INA vertices to be larger than those of the ACT vertices. To achieve this we temporarily add n to the serial number of each INA vertex.

Each ACT vertex v scans its neighbourhood $C(v)$. If $C(v)$ contains a vertex u with a serial number larger than v then v hooks to u, i.e., $parent(v) \leftarrow u$. If such a vertex is not found, then v hooks to the vertex with the largest serial number in $C(C(v))$. This vertex may be v itself. This again takes $O(\log s_{i+1} + \log\log n)$ time.

Each ACT vertex hooks either to itself or to a vertex with a larger serial number. No cycles are thus formed. Suppose now that an ACT vertex v hooked to itself, i.e., $parent(v) = v$, and became the root a tree. We claim that all the vertices in $C(v)$ hooked to v. Let $u \in C(v)$. As $u < v$ and $v \in C(u)$, the vertex u hooks on some vertex $w \in C(u)$. As $w \in C(u) \subseteq C(C(v))$, we get that $w \leq v$ and u does indeed hook to v.

We now show that the size of each tree (which does not correspond to a complete connected component) in the graph after the hooking process is at least s_{i+1}. Each tree of INA vertices was of size s_{i+1} to begin with. Some ACT vertices may have hooked to such trees. This will only increase their size. Consider now a tree T formed entirely from ACT vertices. Let r be the root of this tree. We have shown that all vertices of $C(r)$ will hook on r and thus belong to T. The set $C(r)$ contains at least $s_i^{1/3}$ vertices and the size of each such vertex is at least s_i. The size of T is therefore at least $s_i^{1/3} \cdot s_i = s_{i+1}$ as required.

14 The random walks

A random walk of length s from a vertex $v = v_0$ is obtained by picking, uniformly at random, an edge (v_0, v_1) adjacent to v_0, then an edge (v_1, v_2) adjacent to v_1 and so on, until a path of length s is obtained. A random walk is not necessarily simple and it may return to a previously visited vertex.

Let $G = (V, E)$ be a *simple* connected graph on n vertices, let $v \in V$ be a vertex of G and let $s \leq n$. Barnes and Feige [BF93], proving a conjecture of Linial, showed that the expected number of steps taken by a random walk that starts at v until s distinct vertices are visited is $O(s^3)$. The following Lemma follows immediately from this result using Markov's inequality.

Lemma 14.1 *There exists a constant $a > 0$ such that the probability that a random walk of length s that starts at some vertex v of a simple graph $G = (V, E)$ contains at least $a \cdot s^{1/3}$ distinct vertices, or all the vertices in the connected component of v, is at least $1/2$.*

Proof: Suppose that the expected number of steps needed to encounter s distinct vertices is at most bs^3. Consider a random walk of length s. If the probability that this path encounters at least $(s/2b)^{1/3}$ distinct vertices is less than $1/2$, then the expected length of a path needed to visit $(s/2b)^{1/3}$ vertices is greater than $s/2$. But this is a contradiction as this expectation is at most $b \cdot (s/2b) = s/2$. The Lemma therefore holds with $a = 1/2b$. $\square$

To obtain, with very high probability, neighbourhoods of size $s^{1/3}$ we start from each vertex of the graph $c \log n$ random walks of length $a^3 s$, where a is the constant of Lemma 14.1. The probability that none of the walks initiated from a particular vertex will visit at least $s^{1/3}$ vertices, or all the vertices in that vertex's connected component, is at most n^{-c}. The failure probability of the algorithm can be reduced to $2^{-\Omega(2^{\sqrt{\log n}})}$, which is $o(n^{-c})$, for any $c > 0$, as shown in Section 16. In the sequel we do not mention the explicit constant factors and just say that we perform $\Theta(\log n)$ independent random walks of length $\Theta(s)$ from each vertex.

To implement random walks of length s starting from each ACT vertex we use an $n \times s$ array called *walk*. A similar array was used by Karger *et al.* [KNP92]. We use $L = \Theta(\log n)$ such arrays to implement $\Theta(\log n)$ independent random walks of length s from each vertex.

The entry $walk[v, t]$, for an ACT vertex v and $1 \le t \le s$, is set to a randomly chosen vertex from v's neighbours list $N[v]$. Each entry in the *walk* array is referred to as a *cell.* The value $walk[v, t]$ represents a pointer from the cell $[v, t]$ to the cell $[walk[v, t], t + 1]$. These pointers define trees whose roots are either cells that correspond to INA vertices or cells at time s. If $c = [v, t]$, we say that v is the vertex that corresponds to the cell c.

The matrix *walk* defines a random walk of length s from each ACT vertex v as follows: $u_1 = v$ and $u_t = walk[u_{t-1}, t-1]$, for $1 < t \le s$. Random walks defined for different vertices are *not* independent. This however is not required. Random walks defined by independently chosen *walk* arrays are clearly independent.

Let $T^i_{[v,t]}$ be the tree containing the cell $[v, t]$ of the i-th *walk* array. We would like to define the virtual neighbourhood $C(v)$ of an ACT vertex v as the union of the vertices contained in the trees $T^i_{[v,t]}$ for $1 \le t \le s$ and $1 \le i \le L$. This includes much more than $L = \Theta(\log n)$ independent random walks of length s starting at each ACT vertex and thus all the virtual neighbourhoods will be large enough with very high probability. This definition also satisfies the requirement that $v \in C(u)$ if and only if $u \in C(v)$. The neighbourhoods defined in this way may, however, be *too big* and finding the largest vertex in each neighbourhood would then consume too much time. Luckily, we can show that most of the trees created will not be too big.

Let $[v, t]$ be a cell of the *walk* array. We let $\mathcal{T}_{[v,t]}$ be the random variable giving the size of the tree $T_{[v,t]}$ of the *walk* array that contains the cell $[v, t]$. We let $\mathcal{T}'_{[v,t]}$ be the random variable giving the size of the subtree $T'_{[v,t]}$ of the tree $T_{[v,t]}$ rooted at $[v, t]$. For simplicity, the size of a tree is taken to be the number of edges in it.

Let $p = \langle [v_t, t], [v_{t+1}, t + 1], \ldots, [v_s, s] \rangle$, where $v_t = v$, be a possible path in the *walk* array of length s that corresponds to a graph $G = (V, E)$ with maximal degree d. Let $[u, r]$ be a cell in this *walk* array. We let $E[\mathcal{T}_{[u,r]} \mid p]$ denote the expected value of $\mathcal{T}_{[u,r]}$ given that p is the path chosen from $[v_t, t]$, i.e., the expected value of $\mathcal{T}_{[u,r]}$ given that $walk[v_i, i] = v_{i+1}$ for $t \le i < s$. The conditional expectation $E[\mathcal{T}'_{[u,r]} \mid p]$ is defined similarly.

Lemma 14.2 *Let p be a possible path in the* walk *array and let $[u,r]$ be a cell* not *on this path. Then, $E[\mathcal{T}'_{[u,r]} \mid p] \le \deg(u)\cdot r$.*

Proof: We prove the Lemma by induction on r. If $r = 0$ then the claim is obvious. Suppose therefore that the claim holds for $r-1$. We show that it also holds for r.

Let $w_1, \ldots, w_{\deg(u)}$ be the neighbours of u in G. The subtree $\mathcal{T}'_{[u,r]}$ is composed of the subtrees $\mathcal{T}'_{[w_j,r-1]}$, where $1 \le j \le \deg(u)$, for which $walk[w_j, r-1] = u$. The probability that $walk[w_j, r-1] = u$ is 0 if w_j is an INA vertex or if $[w_j, r-1]$ is on the path p and is $1/\deg(w_j)$ otherwise. We get therefore that

$$E[\mathcal{T}'_{[u,r]} \mid p] \le \sum_{j=1}^{\deg(u)} \frac{1}{\deg(w_j)} \cdot (E[\mathcal{T}'_{[w_j,r-1]} \mid p] + 1)\ .$$

Recall that the size of a tree is taken to be the number of edges in it. Using the induction hypothesis we get that $E[\mathcal{T}'_{[w_j,r-1]} \mid p] \le \deg(w_j)\cdot(r-1)$ and therefore

$$E[\mathcal{T}'_{[u,r]} \mid p] \le \sum_{j=1}^{\deg(u)} \frac{1}{\deg(w_j)} \cdot (\deg(w_j)\cdot(r-1)+1) < \deg(u)\cdot r\ .$$

This completes the proof of the Lemma. It is easy to check that the claim of the Lemma also holds if $[u,r]$ is the first cell on the path p. $\Box$

We are now able to bound the expected size of the trees of the *walk* array.

Lemma 14.3 *Let $[v,t]$ be a cell in a* walk *array of length s that corresponds to a graph $G = (V,E)$ with maximal degree d. Then, $E[\mathcal{T}_{[v,t]}] \le ds^2$.*

Proof: Let $p = \langle [v_t, t], [v_{t+1}, t+1], \ldots, [v_s, s] \rangle$, where $v_t = v$, be a possible path in the *walk* array. This means that $\langle v_t, v_{t+1}, \ldots, v_s \rangle$ is a path in the graph G. We show that the expected size of the tree $\mathcal{T}_{[v,t]}$ given that p is the path chosen from $[v,t]$ (which means that $walk[v_i, i] = v_{i+1}$, for $t \le i < s$) is at most ds^2. As this holds for any such path p, the unconditional expectation is also at most ds^2.

Let $d_i = \deg(v_i)$, for $t \le i \le s$, be the degrees of the vertices on the path p. Let $u_{i,1}, \ldots, u_{i,d_{i+1}-1}$, for $t \le i < s$, be the neighbours of v_{i+1}, other than v_i. It is easy to see that the tree $\mathcal{T}_{[v,t]}$ is composed of the path p, the subtree $\mathcal{T}'_{[v,t]}$ rooted at $[v,t]$, and from the subtrees $\mathcal{T}'_{[u_{i,j},i]}$, where $t \le i < s$ and $1 \le j < d_{i+1}$, for which $walk[u_{i,j}, i] = v_{i+1}$. The probability that $walk[u_{i,j}, i] = v_{i+1}$ is at most $1/\deg(u_{i,j})$ (it is 0 if $u_{i,j}$ is INA and $1/\deg(u_{i,j})$ if it is ACT). We get therefore that

$$E[\mathcal{T}_{[v,t]} \mid p] \le (s-t) + E[\mathcal{T}'_{[v,t]} \mid p] + \sum_{i=t}^{s} \sum_{j=1}^{d_{i+1}-1} \frac{1}{\deg(u_{i,j})} \cdot (E[\mathcal{T}'_{[u_{i,j}]} \mid p] + 1)\ .$$

As $[v,t]$ is the first vertex of the path p, we get, by the last remark in the proof of Lemma 14.2 that $E[\mathcal{T}'_{[v,t]}|p] = E[\mathcal{T}'_{[v,t]}] \le t\cdot d$. As each of the vertices $u_{i,j}$ is not on the path p, we get, again using Lemma 14.2 that $E[\mathcal{T}'_{[u_{i,j},i]} \mid p] \le \deg(u_{i,j})\cdot i$ and therefore

$$\begin{aligned} E[\mathcal{T}'_{[v,t]} \mid p] \ &\le\ s + d\cdot t + \sum_{i=t}^{s-1} \sum_{j=1}^{d_{i+1}-1} \frac{1}{\deg(u_{i,j})} \cdot (\deg(u_{i,j})\cdot i + 1) \\ &\le\ s + d\cdot t + (d-1)\cdot \sum_{i=t}^{s-1} (i+1) \ \le\ ds^2\ . \end{aligned}$$

This completes the proof of the Lemma. $\Box$

It immediately follows from this Lemma, using Markov's inequality, that the probability that the size of $T^i_{[v,0]}$ is greater than $4ds^2$ is at most $1/4$. The probability that the random walk starting at v will visit less than $s^{1/3}$ vertices is also at most $1/2$. The probability that either one of these bad events will happen is therefore at most $3/4$. Thus, with very high probability, at least one of $\Theta(\log n)$ trees $T^i_{[v,0]}$ that correspond to an ACT vertex v will contain cells corresponding to at least $s^{1/3}$ different vertices, yet its total size will be at most $4s^2d$. Trees whose size is at most $4s^2d$ are said to be *small enough*. The largest vertex contained in a small enough tree can be found in $O(\log s + \log d)$ time. The virtual neighbourhood of an ACT vertex v is defined to be the union of all the small enough trees that contain a cell that corresponds to v.

15 Implementation details

Each ACT vertex of the graph has at least $\Theta(s \log n)$ processors allocated to it. The maximum degree of the STAR graph is at most d. We are to simulate $\Theta(\log n)$ independent random walks of length s out of each vertex. Instead of building $\Theta(\log n)$ separate *walk* arrays of length s, we construct one *walk* array of length $s' = \Theta(s \log n)$. We do not choose pointers however from cells that correspond to times $s, 2s, \ldots, s'$. The $\Theta(\log n)$ portions of this long *walk* array are therefore independent as required.

The implementation details of the random walks and the subsequent hookings are quite intricate. We describe at first an implementation that needs a very large workspace. Later, we describe a way of obtaining an implementation with an almost linear space requirement.

The *walk* array used to implement the random walks is of size $n \times s'$. Each row of the original graph as a row allocated to it. Rows that correspond to DONE vertices are not in use. Each entry of *walk* is referred to as a *cell*. The entry $walk[v,t]$, where v is an ACT vertex and $1 \le t < s'$ is set to contain a pointer to the cell $[u, t+1]$ of the array where u is chosen uniformly at random among the neighbours of v. Each cell $[v,t]$, for $1 < t \le s'$, to which at least one other cell points to, has a pointer to an array $prev[v,t]$ that contains a list of the cells in the $t-1$-th layer that point to the cell $[v,t]$.

The *walk* array is constructed in the following way. At the first, the degree $\deg(v)$ of each ACT vertex is distributed to the s' processors allocated to it. This takes $O(\log s') = O(\log s + \log\log n)$ time. The processor allocated to the cell $walk[v,t]$ then puts in that cell a pair (t,x) where x is selected, uniformly at random, from $\{1, 2, \ldots, \deg(v)\}$. The row that corresponds to each ACT vertex is then sorted. This again takes $O(\log s')$ time. It is then easy to replace each number x with the name of the x-th neighbour of x. This again takes $O(\log s')$ time. As a result of the sorting we also obtain arrays $times[v,u]$ that gives, for every ACT vertex v and each one of its neighbours u, the times in which v chose u.

The arrays $prev[v,t]$, for $1 < t \le s'$, that correspond to a vertex v are then obtained in the following way. Let $u_1, \ldots, u_r$ be the neighbours of v. The arrays $times[u_i, v]$, for $1 \le i \le r$, are concatenated and then sorted. The length of this concatenated array is at most ds' and the sorting therefore takes only $O(\log(ds')) = O(\log d + \log s + \log\log n)$ time. The arrays $prev[v,t]$, for $1 < t \le s'$ are obtained as a result.

Each cell in the *walk* array in a row of an ACT vertex has a processor allocated to it. Each such processor also allocates itself to unique cells in each one of the arrays *times* and *prev*. Each cell of these arrays has a processor allocated to it. Each processor is thus allocated to a fixed number of tasks. The processors allocated to the arrays *times* and *prev* and the processors allocated to the edges are the processors that perform the concatenations and sortings described above. The concatenations of the arrays $times[u_i, v]$, for $1 \le i \le r$, take place in a workspace allocated to the vertex v.

Cells of the *walk* array in rows that correspond to INA vertices do not initially have processors allocated to them. A cell that corresponds to an INA vertex does not point to any other cell of the *walk* array. Other cells in the *walk* array may however be pointing to it, in which case it would have a non-empty *prev* array. Each cell of this *prev* array will have a processor allocated to it although initially no processor was allocated to the corresponding cell of the *walk* array. During the construction of the *prev* arrays we also construct for each INA vertex v an array $used[v]$ that contain all the cells in the the row of v that are pointed to. Each such cell $[v,t]$ that appears in this list has a pointer to the next cell in this list.

The pointers of the *walk* array define a forest. The roots of the trees in this forest are cells that correspond to INA vertices and cells that correspond to times $s, 2s, \ldots, s'$. Using the *prev* arrays we can easily define an Euler tour in each one of the trees of this forest.

Each vertex v has a serial number in the range $\{1,2,\ldots,n\}$. While performing the random walks, the serial numbers of the INA vertices are temporarily increased by n. This makes the serial numbers of all the INA vertices larger than the serial numbers of all the ACT vertices. We say that $v < u$ if the serial number of v is smaller than the serial number of u. We also define a lexicographic ordering on the cells of the *walk* array. We say that $[v_1,t_1] < [v_2,t_2]$ if and only if $v_1 < v_2$ or $v_1 = v_2$ and $t_1 < t_2$. If $c = [v,t]$ is a cell, we let $ver(c) = v$ be the vertex to which this cell corresponds. Each tree of the forest defined by the array *walk* contains a unique maximal element. This maximal element functions in certain computations as the root of the tree.

By performing the first $O(\log(ds^2))$ iteration of the naive list ranking algorithm on the Euler tour of each tree we can identify all the small enough trees, i.e., the trees of size at most ds^2. In each such tree we can then find the maximal cell and distribute it to all the cells of the tree. Let $max_1[v,t]$ be the maximal cell in the tree $T_{[v,t]}$ that contains the cell $[v,t]$. If the tree $T_{[v,t]}$ is not small enough we let $max_1[v,t] = 0$. These operations take only $O(\log(ds^2))$ time.

For each ACT vertex v we then compute $max_1[v] = \max_{1\le t\le s'} max_1[v,t]$ and distribute it to all the cells $[v,t]$ in v's row. This takes only $O(\log s')$ time.

If v is an ACT vertex and $u = ver(max_1[v]) > v$ then v is hooked to u by setting $parent[v] \leftarrow u$. A slightly more difficult task is the task of constructing arrays $children_1[v]$ that give for each INA or ACT vertex v a list of all the vertices that hooked to it in this first round.

If $max_1[v] = max_1[v,t]$ and $u = ver(max_1[v]) > v$, we consider the cell $[v,t]$ to be responsible for the hooking of v on u. We put a mark to that effect in $[v,t]$. We now compress the Euler tour of each tree and leave in it only the marked cells. This list is associated with the maximal cell of the tree. In this way we obtain for every cell $[v,t]$ a list $children_1[v,t]$ of all the vertices that hooked to v as a result of the tree $T_{[v,t]}$. The list $children_1[v,t]$ can be non-empty only if $[v,t]$ is the maximal cell in its tree. Each such list contains at most ds^2 elements. The list $children_1[v]$ is obtained by concatenating the lists $children_1[v,t]$ for every $1 \le t \le s'$. The length of each such list is at most $ds^2 s' \le ds'^3$. The total length of all these lists however is at most n as each vertex appears in at most one of these lists. This completes the first hooking attempt.

We now compute again the maximum in each tree. The value of each cell $[v,t]$ is now taken to be the the pair $(max_1[v],[v,t])$. These pairs are ordered lexicographically. We let $max_2[v,t]$ be the maximal pair found in the tree $T_{[v,t]}$, if this tree is small enough. If the tree $T_{[v,t]}$ is not small enough we again let $max_2[v,t] = 0$. All these values can again be found in $O(\log(ds^2))$ time. For every ACT vertex v we then let $max_2[v] = \max_{1\le t\le s'} max[v,t]$ and we distribute this values to all the cells in v's row.

Suppose now that v is an ACT vertex and that $max_1[v] = [u_0,t_0]$ and $max_2[v] = ([u_1,t_1],[u_2,t_2])$. Note that u_0 is the largest vertex in the virtual neighbourhood $C(v)$ of v and that u_1 is the largest vertex in $C(C(v))$. The vertex u_1 belongs to $C(C(v))$ as $u_1 \in C(u_2)$ and $u_2 \in C(v)$. If $u_1 > v = u_0$, then v is hooked on u_1. Again, we build for every vertex v an array $children_2[v]$ that contains a list of all the vertices that hooked to v in the second hooking attempt. This is done in a way similar to the way used to construct the arrays $children_1[v]$. The length of an array $vertices_2[v]$ is at most $d^2 s'^6$.

Finally, the array $children[v]$ for an ACT vertex v is obtained by concatenating the arrays $children_1[v]$ and $children_2[v]$. The array $children[v]$ of an INA vertex is obtained by appending the arrays $children_1[v]$ and $children_2[v]$ to the existing $children[v]$ array. All these operations require only $O(\log d + \log s') = O(\log d + \log s + \log\log n)$ time.

We now discuss the space requirements of the algorithm. The above implementation requires that a very large block of memory be initially allocated to every vertex of the graph, although most of this memory will not be used. As stated, the space requirements of the algorithm are horrendous. For each vertex v, for example, we should preallocate a region of memory large enough to hold the array $children[v]$ which can potentially be of size $s_i^2 s_{i+1}^6$ and in the final iteration $s_{i+1} \geq n$. The space requirement of the algorithm may be easily reduced however to $O(n^{1+\epsilon})$, for every $\epsilon > 0$, by stopping the algorithm when s_i reaches, say, $n^{\epsilon/8}$. The space requirement up to that point is only $O(n^{1+\epsilon})$. As there are only a constant number of iterations left till the end of the algorithm, we can now use an $O(\log n)$ prefix sums algorithm to allocate the memory required by each vertex in each iteration.

16 An easy $m+n$ processors algorithm

In this section we describe a simple way of reducing the number of processors required by the second stage of our algorithm from $O((m+n)\log n)$ to $O(m+n)$. Although this reduction is not required for the purpose of obtaining our optimal $O(\log n)$ time connected components algorithm, we think that it is of interest. In particular, it supplies a relatively simple algorithm for finding connected components in $O(\log n)$ time using $m+n$ processors without using the techniques developed in the first part of this paper. The technique used to reduce the number of processors can also be used to reduce the failure probability of the algorithm.

To reduce the number of processors required from $O((m+n)\log n)$ to $O(m+n)$, we start the algorithm with $k' = O(\log\log n)$ iterations in which we use the actual neighbourhoods of the STAR vertices of the graph and do not perform any random walks. Note that the size of the actual neighbourhood of a STAR vertex that does not correspond to a complete connected component is at least two. During these iterations we use the exponential sequence $s_0 = 2$ and $s_{i+1} = 2s_i$ instead of the doubly exponential sequence described earlier. These k' iterations require only $O(\sum_{i=0}^{k'} \log s_i) = O((\log\log n)^2)$ time. After these k' iterations $s_{k'} = \Omega((\log n)^2)$. In the i-th iteration, for $i > k'$, we then start $\Theta(\log n)$ random walks of length $\Theta(s_i/\log n) \geq \Theta(s_i^{1/2})$ from each ACT vertex. We have enough processors for this as each ACT vertex is of size at least s_i and has at least s_i processors allocated to it. The virtual neighbourhood of each vertex will be, with very high probability, of size at least $(s_i^{1/2})^{1/3} = s_i^{1/6}$. We will henceforth continue with the sequence $s_{i+1} = s_i^{7/6}$.

To reduce the failure probability we start with $k' = 2\sqrt{\log n}$ iterations in which we use the actual neighbourhoods of the STAR vertices. These iterations take $O(\log n)$ time. After these k' iterations $s_{k'} \geq 2^{2\sqrt{\log n}}$. In the i-th iteration, for $i > k'$, we start $\Theta(2^{\sqrt{\log n}})$ random walks of length $\Theta(s_i/2^{\sqrt{\log n}}) \geq \Theta(s_i^{1/2})$ from each ACT vertex. Again, we have enough processors as each ACT vertex is of size at least s_i and has at least s_i processors allocated to it. The probability that the virtual neighbourhood of an ACT vertex will be of size less than $s_i^{1/6}$ is now at most $2^{-\Omega(2^{\sqrt{\log n}})}$.

17 Concluding remarks

Although the connectivity algorithm we present is optimal, there are certain ways in which it could be improved. Our algorithm is quite complicated and it would be interesting to obtain a simpler one with the same performance. A major improvement would be to obtain a *deterministic* algorithm that has the same performance. This, at present, seems like a very difficult task as no deterministic optimal algorithm for finding the connected components is yet known even in the much stronger CRCW PRAM model. A slightly less ambitious, yet interesting, task is to obtain a deterministic $O(\log n)$ time EREW PRAM algorithm for finding connected components that may perform more than linear work.

An interesting feature of our algorithm, shared also by the algorithm of Karger *et al.* [KNP92], is that it finds the connected components without finding a spanning forest. In a recent work [HZ96], we have been able to obtain a version of the algorithm presented here that does find a spanning forest.

A closely related problem to the problem of finding a spanning forest is the problem of finding a minimum spanning forest. Karger [Kar94] obtained a randomised EREW PRAM algorithm that finds a minimum spanning forest of a graph with n vertices and m edges in $O(\log n)$ time using $O(m/\log n + n^{1+\epsilon})$ processors, for any $\epsilon > 0$. Cole, Klein and Tarjan [CKT94],[CKT96] recently obtained a parallel version of the recently discovered randomised linear time algorithm for finding minimum spanning trees of Karger, Klein and Tarjan [KKT95]. Their algorithm runs in $O(\log n)$ time in the CRCW PRAM model.

Acknowlegement

The authors would like to thank two anonymous referees for their careful reading of the paper and for their extensive list of comments that helped us to improve the presentation and the accuracy of the paper.

References

[AKL+79] R. Aleliunas, R.M. Karp, R.J. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal sequences and the complexity of maze problems. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico*, pages 218–223, 1979.

[AM91] R.J. Anderson and G.L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859–868, 1991.

[AS87] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C-36:1258–1263, 1987.

[AS92] N. Alon and J.H. Spencer. *The probabilistic method.* Wiley, 1992.

[BF93] G. Barnes and U. Feige. Short random walks on graphs. In *Proceedings of the 25rd Annual ACM Symposium on Theory of Computing, San Diego, California*, pages 728–737, 1993.

[Bre74] R.P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21:201–208, 1974.

[CDR86] S. Cook, C. Dwork, and R. Reischuk. Upper and lower bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15:87–97, 1986.

[CKT94] R. Cole, P.N. Klein, and R.E. Tarjan. A linear-work parallel algorithm for finding minimum spanning trees. In *Proceedings of the 6th Annual ACM Symposium on Parallel algorithms and architectures, Cape May, New Jersey*, pages 11–15, 1994.

[CKT96] R. Cole, P.N. Klein, and R.E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proceedings of the 8th Annual ACM Symposium on Parallel algorithms and architectures, Padua, Italy*, 1996.

[CL95] K.W. Chong and T.W. Lam. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. *Journal of Algorithms*, 18:378–402, 1995.

[CLC82] F.Y. Chin, J. Lam, and I.N. Chen. Efficient parallel algorithms for some graph problems. *Communications of the ACM*, 25(9):659–665, 1982.

[Col88] R. Cole. Parallel merge-sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[CV88] R. Cole and U. Vishkin. Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal on Computing*, 17(1):128–142, 1988.

[CV89] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 81:334–352, 1989.

[CV91] R. Cole and U. Vishkin. Approximate parallel scheduling. Part II: Applications to logarithmic-time optimal graph algorithms. *Information and Computation*, 92:1–47, 1991.

[DKR94] M. Dietzfelbinger, M. Kutyłowski, and R. Reischuk. Exact lower time bounds for computing boolean functions on CREW PRAMs. *Journal of Computer and System Sciences*, 48:231–254, 1994.

[Gaz91] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM Journal on Computing*, 20(6):1046–1067, 1991.

[GMV91] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico*, pages 698–710, 1991.

[Goo91] M.T. Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico*, pages 711–722, 1991.

[Gre94] J. Greiner. A comparison of parallel algorithms for connected components. In *Proceedings of the 6th Annual ACM Symposium on Parallel algorithms and architectures, Cape May, New Jersey*, pages 16–25, 1994.

[Hag90] T. Hagerup. Optimal parallel algorithms on planar graphs. *Information and Computation*, 84:71–96, 1990.

[Hag92] T. Hagerup. The log-star revolution. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science, Vol. 577, pages 259–278. Springer, 1992.

[Hag93] T. Hagerup. Fast deterministic processor allocation. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms, Austin, Texas*, pages 1–10, 1993.

[HCS79] D.S. Hirschberg, A.K. Chandra, and D.V. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[HZ94] S. Halperin and U. Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. In *Proceedings of the 6th Annual ACM Symposium on Parallel algorithms and architectures, Cape May, New Jersey*, pages 1–10, 1994.

[HZ96] S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests and for other basic graph connectivity problems. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, Atlanta, Georgia*, pages 438–447, 1996.

[IK94] K. Iwama and Y. Kambayashi. A simpler parallel algorithm for graph connectivity. *Journal of Algorithms*, 16:190–217, 1994.

[JáJ92] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[JM91] D.B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2}|V|)$ parallel time for the CREW PRAM. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, San Juan, Puerto Rico*, pages 688–697, 1991.

[JM92] D.B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. In *Proceedings of the 4th Annual ACM Symposium on Parallel algorithms and architectures, San Diego, California*, pages 363–372, 1992.

[Kar93] D.R. Karger. Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms, Austin, Texas*, pages 21–30, 1993.

[Kar94] D.R. Karger. *Random Sampling in Graph Optimization Problems.* PhD thesis, Department of Computer Science, Stanford University, 1994.

[KKT95] D.R. Karger, P.N. Klein, and R.E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42:321–328, 1995.

[KNP92] D.R. Karger, N. Nisan, and M. Parnas. Fast connected components algorithms for the EREW PRAM. In *Proceedings of the 4th Annual ACM Symposium on Parallel algorithms and architectures, San Diego, California*, pages 373–381, 1992.

[KR90] R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, chapter 17, pages 869–932. Elsevier and The MIT Press, 1990.

[KT94] P.N. Klein and R.E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing, Montréal, Canada*, pages 9–15, 1994.

[LF80] R.E. Ladner and M.J. Fischer. Parallel prefix computation. *Journal of the ACM*, 27:831–838, 1980.

[MR89] G. L. Miller and J.H. Reif. Parallel tree contraction, Part 1: Fundamentals. In S. Micali, editor, *Advances in Computing Research, Volume 5, Randomness and Computation*, pages 47–72. JAI Press, 1989.

[NSW92] N. Nisan, E. Szemerédi, and A. Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science, Pittsburgh, Pennsylvania*, pages 24–29, 1992.

[Rad94] T. Radzik. Computing connected componnets on EREW PRAM. Technical Report 94/02, King's College London, 1994.

[Rei84] J.H. Reif. Optimal parallel algorithms for graph connectivity. Technical Report TR-08-84, Center for Computing Research, Harvard University, 1984.

[SV82] Y. Shiloach and U. Vishkin. An $O(\log\ n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.