Once again: Connected Components on a CREW-PRAM

Kai Werther*
Institut für Informatik der Universität Bonn
Römerstr. 164, D-53117 Bonn
Germany

Abstract

We present an algorithm to compute the connected components of a graph in parallel time $\mathcal{O}(\log|V| \log\log|V|)$ on a CREW-PRAM with a linear number of processors. This closes the gap to the best known (and trivial) lower bound of $\Omega(\log|V|)$ from previously $\mathcal{O}(\log^{1/2}|V|)$ to $\mathcal{O}(\log\log|V|)$.

1 Introduction

One of the basic problems in graph theory is to decide whether a given undirected graph G is connected and to compute its maximally connected subgraphs, the so called connected components of G. More formally, given a graph $G = (V, E)$ with n vertices and m edges we want to compute a function $\lambda : V \to V$ such that $\lambda(v) = \lambda(u)$ if and only if u and v lie in the same connected component of G. The function λ can be modeled by a directed graph $C = (V, E')$ with edge set $E' = \{(v, \lambda(v)) \mid v \in V\}$, a so called pointer graph.
There exists a simple linear time sequential algorithm for this problem (see [Ta 72]), the depth-first search. Unfortunately depth-first search seems hard to parallelize. Reif proved in [Re 85] that it is P-hard to compute the lexicographically first depth-first search tree. Although a randomized NC-algorithm was given in [AAK 90], no deterministic NC-algorithm is known for this problem.
All parallel algorithms for computing the connected components follow more or less the approach of Hirschberg ([Hi 76],[HCS 79]) that can be described as follows:
"For all vertices of G in parallel some edge from its edge list is chosen. This gives a pointer graph consisting of a disjoint union V' of pseudo trees. Vertices lying in one pseudo tree are identified to belong to the same connected component of G. Finally a new graph G' with vertex set V' is created. Two vertices in G' are connected if there is an edge between the corresponding trees in G. Then this procedure is iterated with the graph G'. The iteration stops with a graph G'' having no edges. The vertices of G'' respresent the connected components of the input graph G."
The parallel algorithm of [HCS 79] works for the CREW-PRAM and — under the assumption of unit cost (which we will keep throughout the paper) — has a running time of $O(\log^2 n)$ with n^2 processors (here the input was assumed to be the adjacency matrix of G). The number of processors was reduced to $n^2/\log^2 n$ using Brent's theorem (c.f. [Br 74]) by Chin et. al. ([CLC 82]). For the CRCW-PRAM the best known algorithm

*email: kai@cs.uni-bonn.de

works in time $\mathcal{O}(\log n)$ ([SV 82]) with an optimal number of $(n+m)/\log n$ processors ([CV 86], [Ga 86]). The running times of all these algorithm differ by a factor of $\log n$ from the best known (and trivial) lower bounds of $\Omega(\log n)$ for the CREW-PRAM and $\Omega(1)$ for the CRCW-PRAM.
Johnson and Metaxas [JM 91] gave the first algorithm of time complexity $o(\log^2 n)$ for the CREW-PRAM. Their algorithm, that runs in time $\mathcal{O}(\log^{1.5} n)$ with a linear number of processors, was a breakthrough, since former it was widely believed that no algorithm of time complexity $o(\log^2 n)$ would exist (c.f. [KR 90]).
In this paper we develop their algorithm further. Familarity with the paper of Johnson and Metaxas is therefore useful as we will only sketch the necessary algorithmic techniques already contained in there. We will concentrate on the new idea that allows a refinement of the growth control scheduling of Johnson & Metaxas. The central improvement is the observation that to guarantee the growth of a "small" component by a "small" factor only a "small" part of the edge list of the vertex representing that component has to be considered. This allows an arbitrary deep and efficient nesting of layers of computation. In this light we get a uniform description of a sequence of algorithms; the first algorithm of this sequence is a variant of the algorithm of [HCS 79], the second is the one described in [JM 91]. The ℓ-th algorithm of this sequence has a parallel running time of $\mathcal{O}(\ell \log n (\log n)^{1/\ell})$, thus the $\log\log n$-th algorithm with a running time of $\mathcal{O}(\log n \log\log n)$ is the best algorithm of this sequence.
Many algorithms for other graph theoretic problems including Euler tours [AV 84] [AIS 84], ear decompostion [MSV 86], biconnectivity, rely on a connectivity algorithm, so their parallel time complexity also decreases correspondingly. Moreover, the new technique might be directly applicable in other cases as well.
The paper is organized as follows. Section 2 contains a precise definition of the problem and some definitions and notations used throughout the paper. Section 3 describes the elementary steps of parallel algorithms for the connectivity problem and shows how to implement them efficiently. Before describing the general algorithm in Section 5, we first describe the simpler third algorithm from the sequence in Section 4. From the proof of correctness of this algorithm the correctness of all algorithms of the sequence is immediately clear.

2 Notations and definitions

Given an undirected graph $G=(V,E)$, the task of computing the connected components of G can be formalized to produce a directed graph $C=(V,E')$ in which each vertex has outdegree exactly one (the set of edges can thus be interpreted as a function $\lambda : V \rightarrow V$) with the property that $\lambda(u)=\lambda(v)$ if and only if u and v belong to the same connected component of G. In the following we freely switch between λ and C.
The input graph G is assumed to be given in edge list representation in the following way: The edge lists are doubly linked lists and each edge $e=(u,v)$ appearing in the edge list of u has a pointer to its *twin edge* (v,u) appearing in the edge list of v. The digraph C (resp. the function λ) is represented by one pointer for each vertex and initially each vertex points to itself. Vertices in the image of λ are called *active*; non-active vertices are called *passive*. In the beginning all vertices are active. An active vertex with empty edge list is called *finished*. It represents a whole connected component of the input graph G which we also call finished in this context.

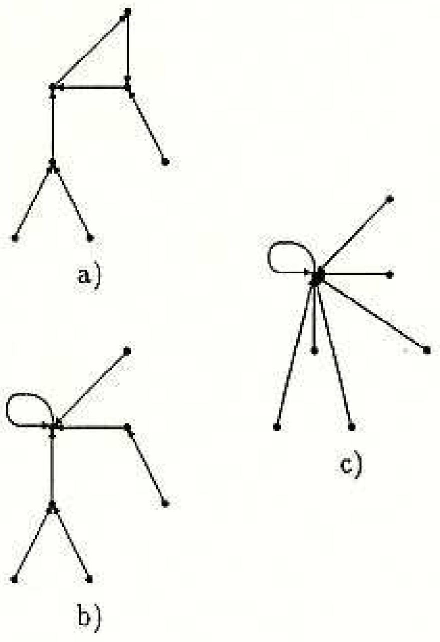

Figure 1: a) pseudo tree, b) rooted tree, c) rooted star

In the course of the algorithm the graphs G and C are modified. To allow an efficient modification of the graph G we reinterpret the data structure of G in dependence of λ in that passive vertices have no neighbours in G independent of their edge list. Note that the properties "active" and "passive" are not evaluated in the algorithm and are solely introduced for a better and clearer description of the algorithm. With this reinterpretation one can pass efficiently the edge list of some vertex to another vertex by simply changing a few pointers. Indeed, even all vertices can do this simultaneously such that large edge lists are constructed.

The general approach of parallel algorithms is to generate a sequence of pointer graphs C_i with the property that if $\lambda_{C_i}(u) = \lambda_{C_i}(v)$ then u and v belong to the same connected component. Furthermore, if $\lambda_{C_i}(u) = \lambda_{C_i}(v)$ then $\lambda_{C_j}(u) = \lambda_{C_j}(v)$ for all $j \geq i$. Eventually, the sequence converges to the output graph C where all vertices in a connected component have the same λ-value.

In general a pointer graph C' is a *pseudo forest*, i.e. a collection of disjoint pseudo trees. A *pseudo tree* is a connected graph containing exactly one cycle (or self loop). The graphs C_i generated by the algorithms have the additional property that the cycle is directed and that all edges are directed towards the cycle.

If the cycle is a self loop, the pseudo tree is called a *rooted tree* and the (unique) vertex v with $\lambda(v) = v$ is called the root. If additionally all vertices have their edges directly to the root ($\forall u : \lambda(u) = v$), the tree is called a rooted star (see Figure 1).

In this terminology the output graph C is a "galaxy," a collection of stars.

Given a pseudo forest C we call the (connected) subgraphs of G that are induced by the connected components of C the C-components of G. Edges between vertices of the same C-component are called *internal*, the edges between two C-components are *parallel* to each other. One (arbitrary) edge from a set of parallel edges is a *useful* edge. All

non-useful edges are redundant and are of no help anymore for the further computation of the connected components.
The diameter of a (directed) graph G is the length of a longest (directed) simple path in G.

3 The building blocks

In the design of parallel algorithms for the connectivity problem three general substeps can be distinguished.

3.1 The hooking step

In the hooking step the pointer graph C is changed as follows: the root v of a pseudo tree (which has $\lambda(v) = v$) selects an vertex u guaranteed to be in the same connected component as v and hooks to it, i.e., it sets $\lambda(v) = u$. Furthermore, the graph G is changed: the edge list of v is inserted into the edge list of u. This substep of the hooking step is also called edge-plugging step.
Johnson & Metaxas propose the following way to accomplish this step efficiently: in parallel each vertex checks whether it is a root and, if this is the case, selects the first edge (say to vertex u) of its edge list and hooks to it. To avoid write conflicts the edge list is inserted after the twin edge in the edge list of u. These manipulations can be done in constant time since only a constant number of pointers have to be reset. The resulting pointer graph is a pseudo forest in which the cycles of the pseudo trees may be arbitrarily large.
Let us summarize.

Observation 1 ([JM 91]) *Given a pair of graphs (G, C) as described above one can compute with n processors in parallel time $\mathcal{O}(1)$ a pair of graphs (G', C') such that if $\lambda_C(v) = v$ and $(v, u) \in E(G)$ is the first edge in the edge list of G then $\lambda_{C'}(v) = u$, otherwise $\lambda_{C'}(v) = \lambda_C(v)$. The edge list of G is changed such that in G' we have*

$$N_{G'}(u) = N_G(u) \cup N_{G'}(v).$$

In the following we denote by "`hook()`" the procedure that executes the hooking step as described in the above observation.
Let us remark that in [HCS 79] instead of the first edge in the edge list the edge to the vertex with the smallest number is selected. This costs $\mathcal{O}(\log n)$ time but has the advantage that each so created pseudo tree has a cycle of length at most two.
Note that through the execution of a hooking step C-components can only grow in their sizes.

3.2 The contracting step

If the pseudo trees are rooted pseudo trees, the standard pointer doubling technique can reduce a pseudo tree with diameter at most d to a rooted star in time $\mathcal{O}(\log d)$. This also holds with minor modifications for pseudo trees with a cycle of length at most a constant. To cope with general pseudo trees, Johnson & Metaxas have defined a set of rules, the so called cycle reducing rules, which can be used to transform an arbitrary pseudo tree with diameter d to a rooted star in time $\mathcal{O}(\log d)$:

Observation 2 ([JM 91]) *Given a pseudo forest C and an integer s one can compute in parallel time $\mathcal{O}(s)$ with n processors a pseudo forest C' with the same connected components such that all pseudo trees of size at most 2^s are rooted stars in C'.*

We will name the procedure that implements Observation 2 "`contract(`s`)`". s is a parameter for the length of the step.
Note that during the hooking step the edge list may be splitted in two circular edge lists. This can be repaired in constant time if one vertex has become the root of the pseudo tree. For details see [JM 91].

3.3 Redundancy removal step

In this step edges that have become redundant during the computation are removed from the graph G. Edges may have become redundant for two reasons:

1. there are edges between two vertices known to be in the same connected component, i.e., vertices with the same λ-value,
2. there are duplicate parallel edges between two pseudo trees, i.e., if there are two edges $e = (u, v)$ and $e' = (u', v')$ with $\lambda(u) = \lambda(u') \neq \lambda(v) = \lambda(v')$ then one of the edges can be removed.

To test the redundancy of an edge each edge $e = (u, v)$ is replaced by the edge $(\lambda(u), \lambda(v))$. Internal edges of C-components corresponding to rooted stars are thus replaced by self loops of the root and can be identified immediately.

3.3.1 The removal of internal edges

In rooted stars, internal edges (i.e. self loops of the root) can easily be removed using the pointer doubling technique on the (linked) edge lists. With s successive pointer doubling steps one can exclude intervals of internal edges of length up to 2^s from the edge list. Since rooted stars of size at most 2^s have at most 2^{2s} internal edges we conclude with the following

Observation 3 ([JM 91]) *Given a pair of graphs G, C and an integer s, one can compute in time $\mathcal{O}(s)$ with $n + m$ processors a graph G' with all self loops of vertices in rooted stars (w.r.t. C) of size $\leq 2^s$ removed.*

The procedure "`internal(`s`)`" implements the above observation. Again, s is the length of the step.

3.3.2 The removal of parallel edges

To remove duplicate edges [JM 91] proceed as follows: the whole edge lists are sorted so that duplicate edges are placed in adjacent cells in the sorted lists. Again using the pointer doubling technique all duplicate edges are removed. This step uses $\mathcal{O}(\log n)$ time. This is an overhead!
What are the purposes of the redundancy removal step? The obvious reason is to get a useful edge in the hooking step. This is accomplished by deleting all internal edges. Small components that have to hook often must have an efficient access to many useful edges. Therefore the edge list has to be purged of redundant parallel edges. Since a

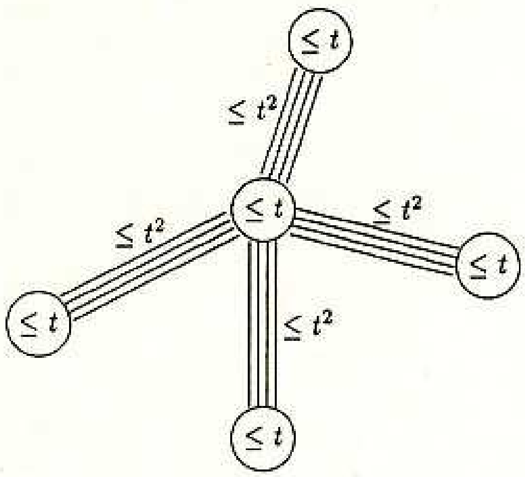

Figure 2: One case of Lemma 4

small C-component has only a few number of edges to some other small C-component, the first part of the edge list either contains many different useful edges or edges to large C-components. The following lemma makes this clear.

Lemma 4 *Let v be the root of a rooted star of size $\leq t$ with no internal edges. Then the edge list of v has less than t^2t' entries or the first t^2t' entries of the edge list contain at least t' different useful edges or one edge to a vertex lying in a C-component of size at least t.*

PROOF: The number of edges between C-components of size at most t is bounded by t^2. If none of the first t^2t' edges in the edge list joins v to a C-component of size more than t, then the first t^2t' edges join v to at least t' different C-components and thus at least t' edges are useful (see Figure 2). Otherwise some edge connects v with a C-component with more than t vertices. ■

Let us call a C-component small if its size is bounded by 2^s and large otherwise. By sorting the first part of the edge list and removing duplicates one can obtain a list of useful edges.

Observation 5 *Given a pair of graphs G, C and integers s, s', such that all small C-components of G are rooted stars and contain no internal edges, one can compute in time $O(s + s')$ a graph G' such that:*
a) the first $2^{s'}$ entries of the edge list of the roots of all small C-components in G are disjoint, or
b) at least one of the edges joins the C-component to a large C-component, or
c) the C-component has fewer than $2^{s'}$ neighbours and to each neighbour there is an edge among the first $2^{s'}$ entries.

Similar to above, the procedure implied by Observation 5 is named "`external`(s, s')".

4 A first improvement: $\mathcal{O}(\log^{4/3} n)$ parallel time

4.1 The first algorithm: Hirschberg et al.

The standard algorithm of [HCS 79] consists of one layer. In this layer a hooking step, contracting and removal (as well internal as external) each of length $\log n$ are performed, and thus all pseudo trees are contracted to rooted stars and all redundant edges are removed. Therefore, after one execution of this layer at least half of the vertices, that were active before the execution of the layer, have become passive or finished, and $\log n$ iterations of this layer suffice to guarantee correctness.

4.2 The second algorithm: Johnson et al.

The algorithm of [JM 91] is built of two layers. The first (deeper) layer (denoted as a "stage" in [JM 91]) consists of a hooking step, contracting and internal removal step of length $\sqrt{\log n}$ each. Here all pseudo trees of size at most $2^{\sqrt{\log n}}$ are reduced to rooted stars and all internal edges of these rooted stars are removed. The second layer (denoted as a "phase" in [JM 91]) consists of $\sqrt{\log n}$ iterations of the first layer and a contracting, internal and external removal step of length $\log n$ afterwards. Thus like in [HCS 79] after the execution of the second layer all redundant edges are removed, but the number of active vertices drops to a fraction of $2^{-\sqrt{\log n}}$. This holds since either the pseudo trees created by the hooking steps consist of at least $2^{\sqrt{\log n}}$ vertices that were active before the execution of the second layer, or in each of the $\sqrt{\log n}$ execution of the first layer they can be reduced to rooted stars. In this case, the number of active vertices in these small trees are at least halved by each execution of the first layer so that after $\sqrt{\log n}$ iterations of the first layer the number of active vertices is reduced to the desired quantity. It follows that $\sqrt{\log n}$ iterations of the second layer guarantee correctness.

4.3 The third algorithm

Lemma 4 removes the drawback of the algorithm of Johnson and Metaxas that either internal edges in small C-components can be removed with very low effort or everything is cleaned up with high effort. Now we can get any **intermediate** level of order in the edge lists of C-components of **corresponding intermediate** size with **corresponding intermediate** effort.

Let us describe how the three-layerd algorithm of the sequence which is given in Figure 3 works and show it correctness. Let $s = \sqrt[3]{\log n}$ and call a C-component small if it contains fewer than 2^s active vertices, medium if it contains fewer than 2^{s^2} active vertices and large otherwise. The purpose of the first layer is to grow small components by a factor of 2. It is easily verified that this can be done by a hooking step, contracting and internal removal steps of length s. After the first layer all small components are contracted and have no internal edges.

The purpose of the second layer is to grow medium C-components by a factor of 2^s. This is done by s iterations of the first layer and afterwards a contracting step of length s^2, an internal removal step of length s^2 and an external removal step with parameters s^2 and s. As above, either the pseudo trees created during the hooking steps have size at least 2^s or they can grow by at least 2^s during the s iterations of the first layer. All pseudo trees of medium size are now contracted and all internal edges are removed. **The execution of**

```
for $i = 1$ to $s$ do
    for $j = 1$ to $s$ do
        for $k = 1$ to $s$ do
            hook()
            contract($s$)
            internal($s$)
        od
        contract($s^2$)
        internal($s^2$)
        external($s^2, s$)
    od
    contract($s^3$)
    internal($s^3$)
    external($s^3, s^2$)
od
```

Figure 3: The third algorithm

the external removal step guarantees that for a next execution of the second layer the first 2^s edges of the edge list of a medium components are useful or one of the edges is an edge to a large component. Thus in the next iteration of the second layer a non useful edge is only encountered if the C-component has grown by a factor of at least 2^s. The execution of the external removal step therefore guarantees the correctness of the next execution of the second layer.
The third layer is like the second layer in [JM 91]. Here, after s iterations of the second layer all C-components have at least 2^{s^2} active vertices or are finished. All C-components are reduced to rooted stars and all redundant edges are removed. Thus s iterations of the third layer again guarantee that all C-components have size $\geq 2^{s^3} = n$ and thus are finished. With the observations of the previous section it is easy to see that the i-th layer has a parallel running time of $\mathcal{O}(s^i)$. Thus the described three-layered algorithm works in time $\mathcal{O}(\log^{4/3} n)$ with a linear number of processors.

5 An $\mathcal{O}(\log n \log\log n)$ algorithm

The above idea generalizes to many-layered algorithms of arbitrary depth. Let us consider an ℓ-layered algorithm and set $s = (\log n)^{\ell^{-1}}$. The structure of the algorithm is best described by a listing, see Figure 4, where r is the number of the layer. Since ℓ might be non-constant we have chosen to write it in a recursive form. The correctness of the algorithm follows as above.
The purpose ot the r-th layer is to guarantee a growth of at least $2^{s^{r-1}}$ for all C-components of size at most 2^{s^r}. As above the correctness ot the r-th layer relies on the fact that the $r-1$-st layer can be consecutively executed as often as required by the use of the external removal step of appropriate length as the last step in the $r-1$-st layer.

```
Connected-Components(1)
   for $i = 1$ to $s$ do
      Layered-Components(1)
   od

Layered-Components($r$)
   if ($r = 1$) then
      hook()
      contract($s$)
      internal($s$)
   else
      for $i = 1$ to $s$ do
         Layered-Components($s, r-1$)
      od
      contract($s^r$)
      internal($s^r$)
      external($s^r, s^{r-1}$)
   endif
```

Figure 4: Recursive formulation of the r-th layer of the general algorithm

The time complexity can be estimated as follows. Let $T_\ell(n,r)$ denote the time complexity of the r-th layer of the ℓ-layered algorithm of the sequence of connected component algorithms and let $T_\ell(n) = sT_\ell(n,\ell)$ be the time complexity of the l-layered algorithm. $T_\ell(n,r)$ can be estimated as follows:

$$T_\ell(n,r) \leq sT_\ell(n,r-1) + cs^r \qquad \text{for } r > 2$$

and $T_\ell(n,1) = cs$ where c is some appropriately chosen constant. It is straight forward to verify that

$$T_\ell(n,r) := crs^r$$

is a solution to the recurrence relation. In particular the time complexity of the ℓ-layered algorithm can be estimated by

$$T_\ell(n) = c\ell \log n (\log n)^{\ell^{-1}}.$$

Some calculation show that the optimal value for ℓ is $\log\log n$. Therefore the time complexity

$$T_{\log\log n}(n) = 2c \log\log n \log n$$

for the $\log\log n$-th algorithm is the best of any algorithm of this family. We summarize this analysis in the following theorem.

Theoem 6 *The connected components of a graph can be computed in time $O(\log n \log\log n)$ with a linear number of processors on a CREW-PRAM without write conflicts. An algorithm with this complexity is given explicitely.*

Let us remark that in the $\log\log n$-th algorithm we have $s = 2$ and the sequence of the lengths of the contracting and removal steps is

$$S = (1,1,2,1,1,2,4,1,1,2,1,1,2,4,8,1,1,2,\ldots).$$

This sequence is the same as the one used by Luby, Sinclair and Zuckerman for the simulation of Las Vegas algorithms [LSZ 93]. Here they show that the above sequence gives a simulation that is optimal up to a constant factor.

Open problems

Is there a superlogarithmic lower bound for the time complexity of computing the connected component of an undirected graph on a CREW-PRAM? I conjecture that a lower bound of $\Omega(\log n \log\log n)$ holds at least for the restricted class of "Hirschberg-type" algorithms.
Much effort has gone in the design of processor optimal algorithms for the connected component problems. Can optimal speed up be achieved with the time bound $\mathcal{O}(\log n \log\log n)$, i.e., can the number of processors be reduced to $(n + m)/(\log n \log\log n)$?

References

[AAK 90] Aggarwal, A., Anderson, R. J., Kao, M.-Y., *Parallel Depth-First Search in General Directed Graphs*, SIAM Journal on Computing **19**(2) (1990), pp. 397–409.

[AV 84] Atallah, M., Vishkin, U., *Finding Euler tours in parallel*, Journal of Computer System Sciences **29** (1984), pp. 330–337.

[AIS 84] Awerbuch, B., Israeli, A., Shiloach, Y., *Finding Euler circuits in logarithmic parallel time*, Proc. 16th ACM STOC (1984), pp. 249–257.

[Br 74] Brent, R. P., *The Parallel Evaluation of General Arithmetic Expression*, Journal of the ACM **21**(2) (1974), pp. 201–206.

[CLC 82] Chin, F., Lam, J., Chen, I., *Efficient parallel algorithms for some graph problems*, Communication of the ACM **25**(9) (1982), pp. 659–665.

[CV 86] Cole, R., Vishkin, U., *Approximate and exact parllel scheduling with applications to list, tree and graph problems*, Proc. 27th IEEE FOCS (1986), pp. 478–491.

[Ga 86] Gazit, H., *An Optiaml Randomized Parallel Algorithm for Finding Connected Components in a Graph*, Proc. 27th IEEE FOCS (1986), pp. 492–501.

[Hi 76] Hirschberg, D., *Parallel Algorithms for the Transitive Closure and the Connect Component Problems*, Proc. 8th ACM STOC (1976), pp. 55–57.

[HCS 79] Hirschberg, D., Chandra, A., Sarwate, D., *Computing Connected Components on Parallel Computers*, Communication of the ACM **22**(8) (1979), pp. 461–464.

[JM 91] Johnson, D. B., Metaxas, P., *Connected Components in $\mathcal{O}(\log^{3/2}|V|)$ Parallel Time for the CREW PRAM*, Proc. 32nd IEEE FOCS (1991), pp. 688–697.

[KR 90] Karp, R. M., Ramachandran, V., *Parallel Algorithms for Shared-Memory Machines*, van Leeuwen, J., editor, Handbook of Theoretical Computer Science, chapter 17, pp. 871–941, Elsevier, 1990.

[LSZ 93] Luby, M., Sinclair, A., Zuckerman, D., *Optimal Speedup of Las Vegas Algorithms*, Information Processing Letters **47**(4) (1993), pp. 173–180.

[MSV 86] Maon, Y., Schieber, B., Vishkin, U., *Parallel ear decomposition search (EDS) and s-t numbering in graphs*, Theoretical Computer Science **47** (1986), pp. 277–298.

[Re 85] Reif, J., *Depth-first search is inherently sequential*, Information Processing Letters **20** (1985), pp. 229–234.

[SV 82] Shiloach, Y., Vishkin, U., *An $O(\log n)$ Parallel Connectivity Algorithm*, Journal of Algorithms **3** (1982), pp. 57–67.

[Ta 72] Tarjan, R., *Depth-first search and linear graph algorithms*, SIAM Journal on Computing **1**(2) (1972), pp. 146–160.