Improved Parallel Algorithms for Finding Connected Components

K.W. Chong* and T.W. Lam†
Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
Email: {kwchong, twlam}@csd.hku.hk

Abstract

Finding the connected components of a graph is a basic computational problem. In recent years, there were several exciting results in breaking the $\log^2 n$-time barrier to finding connected components on parallel machines using shared memory without concurrent-write capability. This paper further presents two new parallel algorithms both using less than $\log^2 n$ time. The merit of the first algorithm is that it uses only a sublinear number of processors, yet retains the time complexity of the fastest existing algorithm. The second algorithm is slightly slower but its work (i.e., the time-processor product) is closer to optimal than all previous algorithms using less than $\log^2 n$ time.

1 Introduction

Given an undirected graph $G = (V, E)$, two vertices $u, v \in V$ are said to be connected if there is a sequence of edges in E linking u and v. The connected component of a vertex v is the set of vertices which are connected to v. The problem of finding the connected components of an undirected graph is often encountered in many applications and is regarded as a fundamental graph problem. This problem can be optimally solved in linear time on a sequential computer using the technique depth-first search or breadth-first search. Unfortunately, these search methods do not admit efficient parallel implementation. Other techniques have been developed to solve this problem in parallel (see the survey in [9, 13]).

This paper is concerned about parallel algorithms for finding connected components on shared-memory machines. The computational model used is the Parallel Random Access Machine (PRAM), which is one of the most popular models for designing parallel algorithms. The PRAM model divides into three main variants in regard to the ability of the processors to concurrently access the shared memory: CRCW, which allows concurrent read and concurrent write; CREW, which allows concurrent read but exclusive write; and EREW, the weakest variant of PRAM, which only allows exclusive read and exclusive write. Details of the model can be found in JáJá's book [9] or the survey by Karp and Ramachandran [13]. The CRCW model is considered much more powerful than the other two. Simulation of a CRCW algorithm on a CREW or EREW PRAM slows down the running time by a factor of $\log n$. The algorithms in this paper are designed for the EREW PRAM.

The performance of a parallel algorithm is

*Research was supported in part by a Postgraduate Studentship and CRCG grant 335/065/0038 of the University of Hong Kong.

†Research was supported in part by CRCG grants 335/065/0038 and 335/065/0039 of the University of Hong Kong.

0-7803-2018-2/95/$4.00 © 1995 IEEE

measured by its running time and work. The work of a parallel algorithm is defined to be the product of the time and the number of processors required, it reflects the total number of operations carried out by all processors. Consider any problem whose fastest sequential algorithm has a time complexity $T(n)$. A parallel algorithm for this problem is said to be work-optimal if its work is $O(T(n))$. In most cases, we are only interested in those work-optimal algorithms running in $\log^{O(1)} n$ time.

Hirschberg, Chandra, and Sarwate [6] are the first to give a parallel algorithm for finding the connected components of an undirected graph in $O(\log^2 n)$ time using $n^2/\log n$ CREW processors, where n and m denote the number of vertices and edges of the graph respectively. The work of this parallel algorithm is $O(n^2 \log n)$. A refinement by Chin, Lam, and Chen [3] improves this algorithm to use $n^2/\log^2 n$ CREW processors with the same time complexity. That is, the work is reduced to $O(n^2)$. This algorithm is work-optimal when the input graph contains $\Omega(n^2)$ edges. A few years later, the processor requirement was further improved by Han and Wagner [7] to $m/\log^2 n + n/\log n$ and the work becomes $O(m + n \log n)$.

Finding connected components in $o(\log^2 n)$ time[1] on the CREW or EREW PRAM had been an open problem for almost a decade [13]. The breakthrough was eventually due to Johnson and Metaxas, who showed an $O(\log^{1.5} n)$ time algorithm using $n + m$ processors on a CREW PRAM [10] or EREW PRAM [11]. Working independently, Karger *et al.*[12] and Nisan *et al.*[14] have also achieved the time complexity $O(\log^{1.5} n)$. A faster algorithm that requires $O(\log n \log\log n)$ time using $n + m$ EREW processors was later given by Chong and Lam [4]. The work of this EREW algorithm is $O((n + m) \log n \log\log n)$.

Note that all previous algorithms that can find connected components in $o(\log^2 n)$ time are not work-optimal. It was not known how to solve the problem in $O(\log n \log\log n)$ time using a sublinear number of processors. Another related open problem was whether $(n+m)/\log n$ processors are sufficient to find connected components in $o(\log^2 n)$ time.

As a matter of fact, the algorithms in [10, 11, 12, 4] all need to repeatedly invoke some fast sorting algorithms such as Cole's Parallel Merge Sort [5], which can sort n numbers in $O(\log n)$ time using n processors, and which incurs $O(n \log n)$ work in total. Each time the sorting is applied on integers lying in the range $[1, n]$. Since sorting n integers in the range $[1, n]$ can be done in $\Theta(n)$ time sequentially, Cole's Parallel Merge Sort is not work-optimal in this case, and this explains why the work of these connected-components algorithms still have a distance from linear. Obviously, one may think of those parallel algorithms designed for sorting integers in a small range, yet the most work-efficient algorithm known for integer sorting, given by Albers and Hagerup [1], runs in $O(\log^{1.5} n(\log\log n)^{0.5})$ time using $n/\log n$ EREW processors. If this integer sorting algorithm is used, though the work may be improved, we apparently would obtain an algorithm much slower than that of [4].

In this paper, we present a parallel algorithm which runs in $O(\log n \log\log n)$ time using $(n + m)/\log\log n$ EREW processors; hence the work is $O((n + m) \log n)$. This is the first algorithm that improves the processor requirement of the fastest existing EREW algorithm to sublinear. The second algorithm to be presented uses $(n + m)/\log n$ EREW processors to solve the problem in

[1] The notation $o(\log^2 n)$ refers to any function $t(n)$ whose growth rate is slower than $\log^2 n$ by more than a constant. More precisely, we mean $\lim_{n\to\infty} \frac{t(n)}{\log^2 n} = 0$. For example, $\log^{1.5} n$ is $o(\log^2 n)$ but neither $\log^2 n - \log n$ nor $\frac{1}{2}\log^2 n$ is.

$O(\log^{1.5} n(\log\log n)^{1.5})$ time. The work incurred is $O((n+m)(\log n)^{0.5}(\log\log n)^{1.5})$, the best among all previous algorithms which run in $o(\log^2 n)$ time. The performance of the second algorithm may be better appreciated if one note that, at present, using $n/\log n$ processors, even sorting n integers in the range $[1, n]$ cannot be achieved in less than $\log^{1.5} n(\log\log n)^{0.5}$ time.

2 Preliminary

Most of the parallel algorithms for finding connected components use an iterative approach: In the first iteration, the vertex set V is partitioned into subsets (each of size at least two) according to some simple rules. Vertices within the same partition must be connected, but two connected vertices may lie in two different partitions. The graph is then contracted to a smaller graph as each partition is represented by one of the vertices within that partition and the adjacency lists of the vertices are combined into one as the adjacency list of the representative vertex. In each subsequent iteration, we work on the representative vertices only and repeat the partitioning and contracting process.

After sufficient iterations, every connected component will be represented by one vertex, and all vertices know the representative vertex of the connected component they belong to. To test whether two vertices are in the same connected component, we simply compare their representative vertices.

The efficiency of such approach depends on two issues: to partition the vertices efficiently and to keep the number of iterations small. For instance, those older algorithms [6, 3, 7] can perform the partitioning in $O(\log n)$ time using $n + m$ EREW processors and the number of iterations can be bounded by $O(\log n)$ [9], hence they can find connected components using $O(\log^2 n)$ time and $O((n + m)\log^2 n)$ work. Intuitively, the number of iterations is determined by the effectiveness of the partitioning process. The number of iterations will be small if, in every iteration, most of the partitions contain a large number of vertices.

The two algorithms presented in this paper are also based on the above approach.

3 Algorithm I

In this section, we show an algorithm to find the connected components of an undirected graph in $O(\log n \log\log n)$ time using $(n + m)/\log\log n$ processors.

3.1 Background

Connect(k): This paper will often make use of a procedure called Connect(k) which was given in [4]: Consider any graph $G = (V, E)$ with n vertices and m edges. Let k be any positive integer bounded by $\log\log n$. Then executing Connect(k) on G will partition the vertices of G in such a way that any vertex $v \in V$ is found in a partition containing at least 2^{2^k} vertices connected to v if v's connected component in G consists of 2^{2^k} or more vertices, otherwise all vertices in v's connected component are put together in v's partition. Connect(k) reports each partition in the form of a rooted tree, and the root will be regarded as the representative of the partition. The time required to execute Connect(k) can be as little as $O(k2^k)$ time provided that $n+m$ processors are available. In particular, if $k = \log\log n$, it takes $O(\log n \log\log n)$ time to execute Connect(k), which can put all vertices in every connected component of G into a rooted tree, but it requires more than $(n+m)/\log\log n$ processors.

In this paper, we will often invoke Connect(k) with $k = \log\log n - \log\log\log n$. This requires $O(\log n)$ time using $n + m$ processors, or, by Brent's scheduling principle [2],

$O(\log n \log\log n)$ time using $(n+m)/\log\log n$ processors. In those cases where m is known to be greater than n, we can simplify the processor complexity as m and $m/\log\log n$, respectively.

Working on a smaller subgraph: As mentioned above, for a graph $G = (V, E)$ with n vertices and m edges, if we use less than a linear number of processors, say, $(n + m)/\log\log n$, executing Connect(k) on G with $k = \log\log n - \log\log\log n$ cannot be done in $O(\log n)$ time. Thus, we will work on a subgraph of G with fewer edges so that the time bound of $O(\log n)$ can be met. Let b be an integer less than n. Define a subgraph $G\langle b\rangle$ with the same vertex set as G as follows:

> Every vertex v chooses b distinct neighbors from its adjacency list, or all if there are less than b neighbors available. $G\langle b\rangle$ includes all edges (u, v) such that u has chosen v or v has chosen u in the previous step.

The subgraph $G\langle b\rangle$ contains at most nb edges. Some vertex in $G\langle b\rangle$ may have more than b neighbors, though. For any vertex $v \in V$, v's connected component in $G\langle b\rangle$ is a subset of that in G. Moreover, if v's connected component in G contains $x < b$ vertices then v's connected component in $G\langle b\rangle$ contains exactly x vertices.

Let $k = \log\log n - \log\log\log n$. Consider the subgraph $G\langle b\rangle$ where $b = 2^{2^k} = n^{\frac{1}{\log\log n}}$. $G\langle b\rangle$ contains at most $n^{1+\frac{1}{\log\log n}}$ edges. Note that $n^{1+\frac{1}{\log\log n}}$ processors are sufficient to execute Connect(k) on $G\langle b\rangle$ in $O(\log n)$ time. For any vertex v of G, if v's connected component in G contains less than b vertices, all vertices in this connected component must be found together after executing Connect(k) on $G\langle b\rangle$; otherwise, v is involved in a connected component with at least $n^{1/\log\log n}$ vertices in both G and $G\langle b\rangle$, executing Connect(k) on $G\langle b\rangle$ would report v in a partition of size at least $n^{\frac{1}{\log\log n}}$ vertices.

Extracting distinct elements from an adjacency list: Our connected component algorithms may contract a given graph G to a smaller multi-graph G'. That is, G' does have fewer vertices than G, but may involve multiple edges between some pair of vertices. Extracting a simple subgraph $G'\langle b\rangle$ from G' is not trivial because the adjacency lists of vertices in G' may contain a lot of duplicate entries. This motivates us to study the following problem.

> Consider any $h \leq n$ linked lists, each composed of integers not necessarily distinct. The total length of these lists, denoted by l, is at most n^2. We want to devise a parallel algorithm using $l/\log\log n$ processors to extract as many as $n^{\frac{1}{\log\log n}}$ elements from each list in $O(\log n)$ time.

A simple solution to this problem is to sort each list in parallel. Identical elements within each list become adjacent elements. Then it is easy to get rid of the redundant elements in each list and then extract the first $n^{\frac{1}{\log\log n}}$ remaining elements from each list. This, however, requires $O(\log n)$ time using l processors [5], or $O(\log n \log\log n)$ time using $l/\log\log n$ processors. A more elaborate solution is as follows: Divide each list into segments; each segment except the last one contains $n^{\frac{1}{\log\log n}} \log n$ consecutive elements. Sort all the segments in parallel and compress identical elements in each segment. This can be done in $O(\log n/\log\log n)$ time using l processors, or $O(\log n)$ time using $l/\log\log n$ processors. If a list has only one segment or contains a segment composed of at least $n^{\frac{1}{\log\log n}}$ distinct elements, then it is done. For all other lists whose segments each contains less than $n^{\frac{1}{\log\log n}}$ distinct elments, their total length must have been shortened by a factor of $\log n$ and is at most $l/\log n$. We can

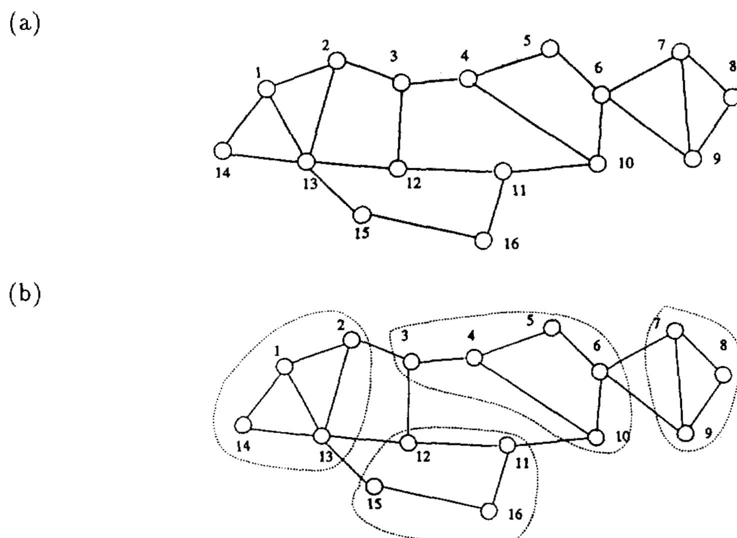

Figure 1: (a) An undirected graph. (b) The vertices are partitioned into groups, each of which are connected. Edges between vertices inside the same partition are internal edges (e.g. edges (1,2), (1,13)). Edges (13,12) and (13,15) are multiple edges between the leftmost and the lower partition. Other multiple edges are (6,7) and (6,9), as well as (3,12) and (11,10).

now sort all the remaining lists and extract all their distinct elements in $O(\log n)$ time using $O(l/\log n)$ processors.

3.2 The algorithm

Given an undirected graph G with n vertices and m edges, we want to find the connected components of G using $(n+m)/\log\log n$ processors. We first consider the case in which the number of processors available, $(n+m)/\log\log n$, is no less than $n^{1+\frac{1}{\log\log n}}$. We will show later how to deal with the case when m is not big enough to guarantee this condition.

The algorithm consists of at most $\log\log n$ phases, each requires $O(\log n)$ time. To ease our discussion, let $G_0 = G$. The input to Phase i, where $0 \leq i < \log\log n$, is a graph G_i, and the output another graph G_{i+1}.

In Phase 0, we extract a subgraph $G_0\langle b\rangle$ from G_0 and execute Connect(k) on $G_0\langle b\rangle$, where $k = \log\log n - \log\log\log n$ and $b = 2^{2^k}$. As mentioned earlier, this requires $O(\log n)$ time using $n^{1+\frac{1}{\log\log n}}$ processors. A partition of size less than b reported by Connect(k) corresponds to a connected component of G_0 and can be reported immediately. For a partition with b or more vertices, we use the root reported by Connect(k) to represent all vertices in the partition and merge the adjacency lists of all vertices in G_0 into a single adjacency list for the root. Note that the new adjacency list may contain internal edges (i.e., edges pointing to vertices inside the same partition) and external edges linking to other partitions. Internal edges can be removed from all the roots' new adjacency lists in $O(\log n)$ time using $m/\log n$ processors. In other words, we contract G_0 to a smaller graph G_1 whose vertices each represents at least b vertices of G_0. G_1 is possibly a multi-graph. See Figure 1.

Next, we proceed to Phase 1. We use the algorithm mentioned earlier to extract a simple subgraph $G_1\langle b\rangle$ from G_1 in $O(\log n)$ time using $O(m/\log\log n)$ processors, and execute Connect(k) to partition the vertices of $G_1\langle b\rangle$. Every connected component of G_1 with less than b vertices is found in a single partition. That means, a connected component of G_0 with less than b^2 vertices can be reported at or before Phase 1. Other partitions containing at least b vertices of G_1 are to be represented by their roots. Thus, G_1 is contracted to a smaller graph G_2 whose vertices each represents at least b^2 vertices of G_0. We repeat these extraction and contraction steps phase after phase until all connected components of G have been reported. Since every connected component of G_0 containing less than b^i or $n^{\frac{i}{\log\log n}}$ vertices can be reported after executing i phases, we conclude that $\log\log n$ phases are sufficient to find the largest connected component of G_0. Each phase can be implemented in $O(\log n)$ time using $(n+m)/\log\log n$ processors.

Preprocessing: At the beginning of the section, we assume that the input graph G is dense and there are sufficient processors. If G has few edges, i.e., $(n+m)/\log\log n < n^{1+\frac{1}{\log\log n}}$, we first execute Connect(k) directly on G with $k = \log\log n - \log\log\log n$. This can be done in $O(\log n \log\log n)$ time using $(n+m)/\log\log n$ processors. G is then contracted to a smaller graph G' with $n' \leq n^{1-\frac{1}{\log\log n}}$ vertices. Note that $n/\log\log n > n'^{\,1+\frac{1}{\log\log n'}}$. We have enough processors to work on G' as if G' is the input. This requires $O(\log n' \log\log n')$ time.

4 Algorithm II

The work incurred by Algorithm I is $O((n+m)\log n)$. To further improve the work, we use a more flexible schedule and a parallel integer sorting algorithm that uses fewer operations. At present, the best of such integer sorting algorithms, devised by Albers and Hagerup [1], can sort x integers drawn from the range $[0, x-1]$ in $O(\log^{1.5} x \log\log^{0.5} x)$ time using $x/\log x$ processors on the EREW PRAM. Note that this algorithm is slower than Cole's $O(\log x)$ time parallel merge sort, but it requires $O(x(\log x \log\log x)^{0.5})$ instead of $x \log x$ operations.

The algorithm presented in this section can find the connected components of an undirected graph G with n vertices and m edges in $O((\log n \log\log n)^{1.5})$ time using $(n+m)/\log n$ processors. We first consider the case where $m/\log n \geq 2n$ and hence more than $2n$ processors are available. Later we will show a simple preprocessing to remove this assumption.

Let $d = m/n\log n \geq 2$. Algorithm II consists of at most $\log(\log_d n) + 1$ phases.[2] Let $G_0 = G$. The input to Phase i, where $0 \leq i \leq \log(\log_d n)$, is a graph G_i with n/d^{2^i-1} vertices, and the output a smaller graph G_{i+1} with $n/d^{2^{i+1}-1}$ vertices. In Phase i, we extract a subgraph G' from G_i, which has the same vertex set as G_i but contains at most $m/\log n$ edges, and then execute Connect(k) with $k = \log\log n$ to find all connected components of G'. Similar to Algorithm I, G_{i+1} is composed of the roots of those relatively large connected components of G' reported by Connect(k). Details are as follows.

Extracting G' from G_i: As G_i may be a multi-graph (i.e., the adjacency list of a vertex may contain duplicate entries), we first apply Albers and Hagerup's algorithm [1] to sort each adjacency list. G_i cannot have more edges than G and the total length of all adjacency lists of G_i is bounded by $m \leq n^2$. Thus, the sorting can be done in $O((\log n)^{1.5}(\log\log n)^{0.5})$ time using $m/\log n$

[2] For example, if $d = 2$, $\log(\log_d n) = \log\log n$; when $d = n^\epsilon$ for some $\epsilon > 0$, $\log(\log_d n)$ will be a constant.

processors. Redundant elements are then removed from each adjacency list and G_i becomes a simple graph. G' is defined to be $G_i(d^{2^i})$. If G_i has at most n/d^{2^i-1} vertices, then G' has at most $(n/d^{2^i-1})d^{2^i}$ or $m/\log n$ edges.

Finding the connected components of G': We execute the procedure Connect(k) with $k = \log\log n$ on G', which has at most n vertices and $m/\log n$ edges. This can be done in $O(\log n \log\log n)$ time using $n + m/\log n$ processors, or simply $m/\log n$ processors (since we assume that $m/\log n \geq n$). Recall that G' satisfies the property that, for any vertex v, if v is connected to less than d^{2^i} vertices in G_i, these vertices including v must be found in a single connected component of G'; otherwise, v's connected component in G' involves at least d^{2^i} vertices. It is easy to prove by induction that each vertex in G_i represents at least d^{2^i-1} vertices that are connected in G_0. In other words, for a connected component of G_0 with no more than $d^{2^i-1}d^{2^i} = d^{2^{i+1}-1}$ vertices, either it has been identified before Phase i or there is a corresponding connected component in G_i involving at most d^{2^i} vertices. At the end of Phase i, all connected components of G_0 which contain less than $d^{2^{i+1}-1}$ vertices must have been found.

Constructing G_{i+1}: G_i has at most n/d^{2^i-1} vertices. The number of connected components of G' which contain d^{2^i} or more vertices is at most $n/d^{2^{i+1}-1}$. We use the roots of such components to represent the vertices of G_i and form a smaller graph G_{i+1}. The details are similar to Algorithm I. The algorithm terminates when G_{i+1} is empty or has only one vertex. In the worst case, the algorithm may run up to Phase $\log(\log_d n)$.

The most time-consuming step in each phase is the integer sorting, which requires $O((\log n)^{1.5}(\log\log n)^{0.5})$ time. As there are at most $\log(\log_d n) + 1$ phases, the time complexity of Algorithm II is $O((\log n \log\log n)^{1.5})$. Note that the denser the input graph, the more efficient Algorithm II is. For instance, if $m > n^{1+\epsilon}\log n$ for some $\epsilon > 0$, then $d = n^{\epsilon}$ and Algorithm II requires only $O(1)$ phases.

Preprocessing: If the input graph G has few edges and $m/\log n < 2n$, we first execute Connect(k) with $k = \log\log\log n + 1$ directly on G and then contract G to a smaller graph G' containing $n' \leq n/2\log n$ vertices. Since there are $(n+m)/\log n \geq 2n'$ processors available, we can now execute Algorithm II on G'.

5 Concluding Remarks

We have presented two improved algorithms for finding the connected components of an undirected graph in the EREW PRAM: one is running in $O(\log n \log\log n)$ time using $(n+m)/\log\log n$ processors while the other takes $O((\log n \log\log n)^{1.5})$ time and $(n+m)/\log n$ processors. The work of these algorithms are $O((n+m)\log n)$ and $O((n+m)(\log n)^{0.5}(\log\log n)^{1.5})$ respectively. It is interesting to know whether these algorithms can be improved to use fewer processors while maintaining the time bound. Using randomization, Halperin and Zwick [8] have lately devised an optimal EREW algorithm which finds connected component in $O(\log n)$ time with high probability using $(n+m)/\log n$ processors. It remains to see whether a deterministic EREW or CREW algorithm using optimal work and less than $\log^2 n$ time (say, $\log^{1+\epsilon} n$ for some $\epsilon < 1$) exists. On the other hand, even we relax the processor requirement to any polynomial (say, n^3), it is still an open problem whether the problem can be solved in time less than $\log n \log\log n$. The ultimate goal is to devise an $O(\log n)$ time algorithm.

References

[1] S. Albers and T. Hagerup, *Improved Parallel Integer Sorting without Concurrent Writing*, Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms, 1992, pp.463-472.

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

[3] F.Y. Chin, J. Lam, and I-N. Chen, *Efficient Parallel Algorithms for some Graph Problems*, Communications of the ACM, 25(1982), pp. 659-665.

[4] K.W. Chong and T.W. Lam, *Finding Connected Components in $O(\log n \log\log n)$ time on the EREW PRAM*, Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms, 1993, pp. 11-20.

[5] R. Cole, *Parallel Merge Sort*, SIAM Journal of Computing, 17(1988), pp. 770-785.

[6] D.S. Hirschberg, A.K. Chandra, and D.V. Sarwate, *Computing Connected Components on Parallel Computers*, Communications of the ACM, 22(1979), pp. 461-464.

[7] Y. Han and R.A. Wagner, *An Efficient and Fast Parallel-Connected Component Algorithm*, Journal of the ACM, 1990, pp. 626-642.

[8] S. Halperin and U. Zwick, *An Optimal Randomized Logarithmic Time Connectivity Algorithm for the EREW PRAM*, Proc. Symposium of Parallel Algorithms and Architectures, 1994, pp. 1-10.

[9] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992, pp. 203-260.

[10] D.B. Johnson and P. Metaxas, *Connected Components in $O(\lg^{3/2}|V|)$ Parallel Time for the CREW PRAM*, Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, 1991, pp. 688-697.

[11] D.B. Johnson and P. Metaxas, *A Parallel Algorithm for Computing Minimum Spanning Trees*, Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures, 1992, pp. 363-372.

[12] D.R. Karger, N. Nisan, and M. Parnas, *Fast Connected Components Algorithms for the EREW PRAM*, Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures, 1992, pp. 373-381.

[13] R.M. Karp and V. Ramachandran, *Parallel Algorithms for Shared-Memory Machines*, Handbook of Theoretical Computer Science, vol A, J. van Leeuwen Ed., MIT Press, Massachusetts, 1990, pp. 869-941.

[14] N. Nisan, E. Szemeredi, and A. Wigderson, *Undirected Connectivity in $O(\log^{1.5} n)$ Space*, Proc. 33rd Annual IEEE Symposium on Foundations of Computer Science, 1992, pp. 24-29.