Ultrafast Randomized Parallel Construction- and Approximation Algorithms for Spanning Forests in Dense Graphs

Anders Dessmark[2], Carsten Dorgerloh[1], Andrzej Lingas[2], and Jürgen Wirtgen[1]

[1] Department of Computer Science, Bonn University, D-53117 Bonn, Germany.
Email: {carsten,wirtgen}@cs.uni-bonn.de.
[2] Department of Computer Science, Lund University, Box 118,S-22100 Lund, Sweden.
Email: {Anders.Dessmark,Andrzej.Lingas}@dna.lth.se.

Abstract. We present a first randomized $\mathcal{O}(\log^{(k)} n)$ time and $\mathcal{O}(n+m)$ work CRCW-PRAM algorithm for finding a spanning forest of an undirected dense graph with n vertices. Furthermore we construct a randomized $\mathcal{O}(\log\log n)$ time and $\mathcal{O}(n \log n)$ work CREW-PRAM algorithm for finding spanning trees in random graphs.

1 Introduction

Finding a spanning forest of an undirected graph $G = (V, E)$ is one of the most basic algorithmic graph problems. A spanning tree of G is a subgraph which is a tree which contains all the vertices of G, a spanning forest is a set of spanning trees, one for each component of the graph. Such objects can be found sequentially in linear time using, for example, a depth-first search or a breadth-first search. In the parallel setting it seems that this problem is far from being trivial. Therefore, the literature on finding spanning trees and forests in parallel has a long and varied history (see e.g. [HZ96] for a summary of PRAM algorithms for those problems). [HZ96] gave the first optimal randomized EREW-PRAM algorithm for those problems which runs in $\mathcal{O}(\log n)$ time using $\mathcal{O}((m+n)/\log n)$ processors. In this work we consider dense graphs, and obtain an optimal randomized arbitrary CRCW-PRAM algorithm for finding a spanning forest in dense graphs which runs in $\mathcal{O}(\log^{(k)} n)$ time using $\mathcal{O}((n+m)/\log^{(k)} n)$ processors, where $\log^{(k)}$ is the k-times iterated logarithm. Formally

$$\log^{(1)} n = \log n \quad \text{and} \quad \log^{(k)} n = \log(\log^{(k-1)} n)\ \forall k > 1$$

We define $\log^* n$ to be the smallest integer k, such that $\log^{(k)} n \leq 2$. The function $\log^* n$ is extremely slow increasing and for instance $\log^* 2^{65536} = 5$.

Ultrafast algorithms for other problems have been presented by Gil, Matias and Vishkin [GMV91]. They used randomization to get $\mathcal{O}(\log^* n)$-time CRCW-PRAM algorithms for a wide range of problems including dictionaries, loadbalancing, hashing and integer chain sorting.

The paper is organized as follows. Section 2 contains some definitions and notations used throughout the paper. In Section 3 we present an arbitrary CRCW-PRAM algorithm for finding a spanning forest in a dense graph. Subject of Section 4 is the construction of a CREW-PRAM algorithm for finding spanning trees in dense random graphs. We introduce a random graph model and present an $\mathcal{O}(\log\log n)$ time CREW-PRAM algorithm for this task, which requires $\mathcal{O}(n \log n)$ work. Finally, we conclude in Section 5 with open problems.

2 Notations and Definitions

Let $G = (V, E)$ be a graph with $n = |V|$ vertices and $m = |E|$ edges. G is called *δ-dense*, if the minimum degree $\delta(G)$ is at least δn (see e.g. [AKK95]). Thus, for dense graphs $m = \Omega(n^2)$. A *dominating set* for G is a subset $V' \subset V$ such that for all $u \in V - V'$ there is a $v \in V'$ for which $\{u, v\} \in E$. Given a set $V' \subset V$, the *subgraph of G induced by V' denoted by $G[V']$* is the graph $G' = (V', E')$, where $E' = \{\{u, v\} \in E : u, v \in V'\}$.
The computational model used in this paper is the *randomized Parallel Random Access Machine* (for short *randomized PRAM*). The randomized *arbitrary CRCW-PRAM* is a synchronized parallel computation model for which simultaneous accesses for both reading and writing are permitted. In case of a concurrent write, an arbitrary processor wins, whereas in the randomized *CREW-PRAM* model simultaneous accesses for writing are forbidden. Furthermore, each processor of a randomized PRAM has access to a random number generator which returns random numbers of $\log n$ bits in constant time. More details of the PRAM models can be found, e.g., in the survey by Karp and Ramachandran [KR88].

3 A CRCW-PRAM Algorithm for Finding Spanning Trees

In this section we show how to find a spanning forest in a δ-dense graph with $\mathcal{O}((n + m)/\log^{(k)} n)$ processors on a randomized arbitrary CRCW-PRAM in $\mathcal{O}(\log^{(k)} n)$ time, where $k \in \mathcal{O}(1)$ can be arbitrary choosen. To achieve this running time we will make heavily use of the following lemma.

Lemma 1. *Let $G = (V, E)$ be a δ-dense graph. A set of*

$$k = \frac{\log(n/\alpha)}{\log(1/(1-\delta))} = O(\log n)$$

randomly chosen vertices R forms a dominating set with probability at least $(1 - \alpha)$.

Proof: The probability, that one particular vertex v will be dominated by one randomly chosen vertex, is at least δ. If we choose k vertices independently,

then the probability that it is not dominated is at most $(1-\delta)^k$. Thus the expected number of not dominated vertices is at most α, because

$$\begin{aligned}(1-\delta)^k n &\leq \alpha \\ n/\alpha &\leq (1/(1-\delta))^k \\ \frac{\log(n/\alpha)}{\log(1/(1-\delta))} &\leq k\end{aligned}$$

By Markov's inequality (cf. [MR95]) we get the lemma. ■

The algorithm first chooses $\mathcal{O}(\log n)$ vertices uniformly at random. This is done as follows. Let each of $2\lceil \log n \rceil$ processors choose a vertex with equal probality and write the number of the chosen vertex in an vector of length $2\lceil \log n \rceil$. Next, to remove duplicates we sort the vector, mark the first occurence of each vertex in the sorted vector and compute the prefix sums. The latter will determine the position of each sorted vertex in a new vector without duplicates. The probability that the number of different vertices chosen is $< \log n$ is smaller than the probability that at least one vertex is choosen by at least 3 processors. Hence, the number of different vertices is in the range $[\log n, 2\log n]$ with probability at least $1 - \mathcal{O}(1/n)$. By using Cole's parallel merge-sort [Col88] and the standard algorithm for prefix sums the removal of duplicates can be done in $\mathcal{O}(\log\log n)$ time using $\mathcal{O}(\log n)$ processors. Denote the set of chosen vertices by R. By Lemma 1, R forms a dominating set with error-probability bounded by an arbitrary small constant.

Now we again compute a dominating set R' in $G[R]$ in exactly the same way as described above and determine the edges of the forest which have one endpoint in R' and the other in $G[R] - R'$. This process is iterated for $k-1$ phases after which we apply the spanning forest algorithm of [HZ96] to the remaining graph.

To extract the graph induced by R we use $\mathcal{O}(\log^2 n)$ processors that in parallel read the information from the adjacency matrix of G and write this in constant time to the adjacency matrix of $G[R]$. In case G is not given in adjacency matrix representation we construct one by using $\mathcal{O}(n+m)$ processors in constant time. By Brent's principle [Bre74] the number of processors can be reduced to $\mathcal{O}((n+m)/\log^{(k)} n)$ if we allow $\mathcal{O}(\log^{(k)} n)$ time.

In order to eliminate the need of extracting $G[R]$ but for the bottom level of the recursion, we can simplify our algorithm to the following one avoiding recursive calls.

Algorithm A_k
Input : a graph G
Output : a forest of G

1. color each vertex u with color 0 and set $ad(u)$ to 0.
2. for $i = 1, ..., k-1$ do
   Let there be n_{i-1} vertices colored with color $i-1$. Choose randomly $\mathcal{O}(\log n_{i-1})$ of them (using the aforementioned method) and recolor them with the color i.

3. retrieve the vertices colored with color $k-1$ using the algorithm [RS85].
4. extract the subgraph of G induced by vertices colored with $k-1$ using the method described before.
5. find a spanning forest T for the (extracted) subgraph induced by vertices colored with $k-1$ using the algorithm [HZ96].
6. mark the vertices and edges of G that are in T and store for each of them the ID of the component they belong to.
7. for $i = k-1, ..., 1$ do
   each processor assigned to an edge whose one endpoint v is both marked and colored with i and the other endpoint u is colored with $i-1$ writes v into the field $ad(u)$.
   for each vertex u with color $i-1$ and a non-zero value of $ad(u)$, T is augmented with the vertex u and the edge $\{u, value(ad(u))\}$, u is marked and store for this vertex the component-ID of $value(ad(u))$.
8. if there is a vertex outside T then go to Step 1.
9. output T.

The functionality of the algorithm is now sufficiently described. However, we have to show that the graph induced by the set of vertices with the highest color does not loose much of its denseness in each recoloring iteration and analyze the complexity of the algorithm in the arbitrary CRCW-PRAM model.

The following lemma guarantees that in each recoloring phase of the algorithm the subgraph induced by the vertices with the highest color is still dense, such that we can compute randomized a dominating set of logarithmic size in constant time (see Lemma 1).

Lemma 2. *Let $G = (V, E)$ be a δ-dense graph and $R \subset V$ be a set with $|R| = \beta \log n$. Then $G[R]$ is $(1-\gamma)\delta$-dense with high probability.*

Proof: We define the random variable X_v for $v \in V$ to be the number of neighbors of v in R. Then $\mathrm{E}[X_v] \geq c \log n$, where $c = \delta\beta$. Now we bound the probability that X_v deviates far from its expectation by applying the Chernoff bound (see ,e.g., [MR95], Theorem 4.2).

$$\begin{aligned}\Pr[X_v < (1-\gamma)\mathrm{E}[X_v]] &< \exp(-\mathrm{E}[X_v]\gamma^2/2) \\ &\leq 1/\exp(c \log n \gamma^2/2) \\ &= 1/2^{c \log n \log e \gamma^2/2} \\ &= 1/n^{c \log e \gamma^2/2} =: p_1(n).\end{aligned}$$

Define Y_v as

$$Y_v := \begin{cases} 0 \text{ if } X_v \geq (1-\gamma)\mathrm{E}[X_v] \\ 1 \text{ otherwise.} \end{cases}$$

and set $Y = \sum_{v \in R} Y_v$. Then the probability that a vertex $v \in R$ has fewer than $(1-\gamma)c \log n$ neighbors in R is at most $\Pr[Y > 1]$. By applying Markov's inequality, we obtain that

$$\Pr[Y \geq 1] \leq \mathrm{E}[Y]$$

$$
\begin{aligned}
&= \sum_{v \in R} \underbrace{\mathrm{E}[Y_v]}_{<p_1(n)} \\
&< p_1(n)\,\beta \log n \\
&= \frac{\beta \log n}{n^{c \log e \gamma^2/2}} =: p_2(n).
\end{aligned}
$$

Thus $G[R]$ is $(1-\gamma)\delta$-dense with probability $1 - p_2(n) = 1 - o(1)$.

■

Lemma 3. *The algorithm A_k can be implemented in expected time $\mathcal{O}(\log^{(k)} n)$ on a randomized arbitrary CRCW PRAM with $\mathcal{O}(n^2/\log^{(k)} n)$ processors using $\mathcal{O}(n^2)$ space.*

PROOF: The partial correctness of the algorithm is obvious. It remains to analyse its complexity, especially the expected time performance.

Steps 1, 6 and 8 take constant time using $\mathcal{O}(n^2)$ processors. Step 2 takes $\mathcal{O}(k)$ time using $\mathcal{O}(n^2)$ processors. It follows from straightforward calculations that the expected size of the set of vertices colored with $k-1$ is $\Theta(\log^{(k-1)} n)$. Hence, the retrieval steps 3 and 4 and the construction of T in step 5 take expected time $\mathcal{O}(\log^{(k)} n)$ by [RS85] and [HZ96], respectively. Step 7 takes $\mathcal{O}(k)$ time and $\mathcal{O}(n^2)$ processors. By Lemma 1 and Lemma 2, the probability that T in step 8 doesn't include all vertices in G is bounded by a positive constant from below. Hence, the expected number of iterations of steps 1 through 8 is $\mathcal{O}(1)$. Now, it is sufficient to apply Brent's principle to meet the bounds in the lemma statement.

■

Observe that the algorithm A_k constructs just a forest which is not necessarily a spanning forest. Note that already the set of all vertices of G (without any edges) forms a forest. First we shall show that the forest constructed by the algorithm A_k is not far away from being a spanning forest - in the sense that the number of components is within a constant multiplicative of the number of components of a spanning forest. Therefore we have a constant-time approximation to a spanning forest in G.

Later we will modify the CRCW-algorithm to get a spanning forest in the same time/work bounds.

Our problem is the following: It depends on how we choose our dominating set R in each step. It can happen, that $G[R]$ looses the connectivity of G. For example $G[R]$ consists of 2 components, although G was connected (see figure 1)

But we can prove the following:

Lemma 4. *If G is δ-dense, then G has at most $1/\delta$ components.*

PROOF: Suppose G has more than $1/\delta$ components. Then there is at least one component with less than δn vertices, which is a contradiction to the δ-denseness. ■

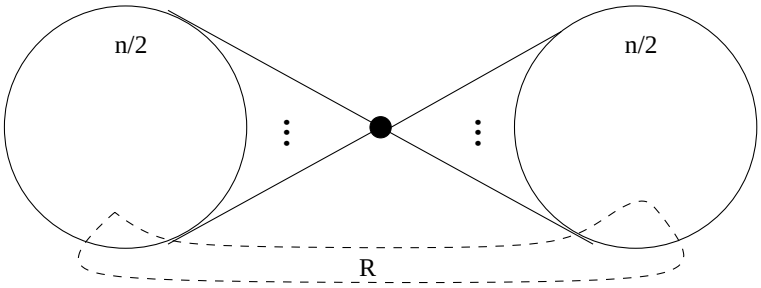

Fig. 1. There is a small separator in G which is not in R

Since in each iteration of A_k the graphs remain reasonable dense, we can guarantee a constant approximation to the number of components in G.

However we can modify the algorithm A_k by deleting the step 9 and inserting the steps $9'$ to $13'$, yielding the algorithm A'_k, which constructs a spanning forest in the same bounds as A_k:

$9'$. Build a constant size matrix C, with rows and columns for each component of T.
$10'$. For each edge $\{u, v\}$ of G, with u in component i and v in component j, write in parallel a 1 and $\{u, v\}$ into $C_{i,j}$ and $C_{j,i}$.
$11'$. Store for each component i the minimum of its own and of the adjacent component's IDs in $s(i)$.
$12'$. Each component i changes its ID to $s(i)$ and adds the edge in $C_{i,s(i)}$ to the component.
$13'$. Output T.

We again summarize this description in the following theorem.

Theorem 5. *Given a dense graph G on n vertices, a spanning forest in G can be found in $\mathcal{O}(\log^{(k)} n)$ time by a randomized arbitrary CRCW-PRAM with $\mathcal{O}(n^2/\log^{(k)} n)$ processors using $\mathcal{O}(n^2)$ space.*

Proof: Observe that the execution of the steps $9'$ to $13'$ take constant time. ■

4 An CREW-PRAM Algorithm for Spanning Trees in Random Graphs

In this section we develop a randomized CREW-PRAM algorithm for finding a spanning tree in a dense random graph in $\mathcal{O}(\log\log n)$ parallel time using $\mathcal{O}(n \log n/\log\log n)$ processors. If no adjacency matrix is given, $\mathcal{O}(n^2/\log\log n)$ processors are required.

The random graph model is as follows. We denote by $G(n,p)$ the random graphs on n vertices, in which each possible edge occurs with probability p (independent of other edges). It is clear, that for a constant p, $G(n,p)$ is almost surely dense. Here *almost surely* means that the statement holds with probability that tends to 1 as n goes to infinity. $G_{n,m}$ is the random graph on n vertices and m edges. Each graph with m edges and vertices numbered 1 through n is equally likely in this model. Its behavior is very similar to the behavior of $G(n,p)$, for $p = \frac{m}{\binom{n}{2}}$.

Erdős and Rènyi gave in [ER60] the following threshold for connectivity in $G_{n,m}$.

Theorem 6 ([ER60],[McD97]). *If $m = \frac{1}{2}n(\log n + c_n)$ and $c_n \to c$ as $n \to \infty$, then*

$$\Pr[G_{n,m} \text{ is connected}] \to e^{-e^{-c}} \quad \text{as } n \to \infty.$$

It follows that if $c_n \to -\infty$ then this probability goes to 0, and if $c_n \to \infty$ then it goes to 1. Therefore, $G(m,p)$ is almost surely connected for constant p. Now it is easy to prove the following lemma.

Lemma 7. *Let $G = G(n,p)$ be a random graph as described above and let p be a constant. If R is a randomly chosen vertex subset of G, then $G[R]$ is almost surely connected.*

To determine a spanning tree in in $G = G(n,p)$ we construct an adjacency matrix of G if none is given and choose randomly $R \subseteq V(G)$ of size $\mathcal{O}(\log n)$ analogously as in Section 3. We can do it on an EREW-PRAM in $\mathcal{O}(\log\log n)$ time using $\mathcal{O}(\log n)$ processors. Now we extract $G[R]$ with a CREW-PRAM in time $\mathcal{O}(1)$, using $\mathcal{O}(\log^2 n)$ processors. We know by Lemma 7 that $G[R]$ is almost surely connected. Using the EREW-PRAM algorithm of Halperin and Zwick [HZ96] we construct a spanning forest in $G[R]$ (If the forest is not a tree we randomly pick a new subset R and repeat the above steps). This takes time $\mathcal{O}(\log\log n)$ with $\mathcal{O}(\log n)$ processors. By Lemma 1 it remains to add the edges of the spanning tree which have one endpoint in $V - R$ and the other in R. For each vertex v in $V - R$ we produce a copy of the vector containing R. Next, on the basis of the copy, we produce a $0-1$ vector indicating which of the members in R are adjacent to v. This can be done in constant time using $\mathcal{O}(n \log n)$ processors or in $\mathcal{O}(\log\log n)$ time using $\mathcal{O}(n \log n / \log\log n)$ processors. Then by using tournament style pairwise comparisons we can pick the leftmost member of R that is adjacent in time $\mathcal{O}(\log\log n)$ using $\mathcal{O}(n \log n / \log\log n)$ processors.

We can summarize the above in the following theorem.

Theorem 8. *For dense random graphs in $G(n,p)$, a spanning tree can be computed in $\mathcal{O}(\log\log n)$ time by a randomized CREW-PRAM with $\mathcal{O}(\frac{n \log n}{\log\log n})$ processors using $\mathcal{O}(n^2)$ space. If no adjacency matrix is given, $\mathcal{O}(n^2/\log\log n)$ processors are required.*

5 Open Problems and Further Research

We have proposed an optimal randomized CRCW-PRAM algorithm, which finds a spanning forest in dense graphs in $\mathcal{O}(\log^{(k)} n)$ time. We can choose k to be an arbitrary large constant. Can we speed up our algorithm to $\mathcal{O}(\log^* n)$ time with the same number processors? Another interesting question would be to derandomize this algorithm.

Moreover we designed an CREW-PRAM algorithm for constructing spanning trees in dense graphs with a very good average case performance. A very fundamental open problem, is the design of efficient CREW-PRAM algorithms for dense graphs, with a significantly sublogarithmic running time, in the worst case model.

References

[AKK95] S. Arora, D. Karger, and M. Karpinski. Polynomial time approximation schemes for dense instances of np-hard problems. In *ACM STOC*, volume 27, pages 284–293, 1995.

[Bre74] Richard P. Brent. The parallel evaluation of general arithmetic expression. *Journal of the ACM*, 21(2):201–206, 1974.

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

[ER60] P. Erdős and A. Rènyi. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci.*, 5:17–61, 1960.

[GMV91] Joseph Gil, Yossi Matias, and Uzi Vishkin. Towards a theory of nearly constant time parallel algorithms. In *IEEE FOCS*, volume 32, pages 698–710, 1991.

[HZ96] Shay Halperin and Uri Zwick. Optimal randomized erew pram algorithms for finding spanning forests and for other basic graph connectivity problems. In *ACM-SIAM SODA*, volume 7, pages 438–447, 1996.

[KR88] Richard M. Karp and Vijaya Ramachandran. A survey of parallel algorithms for shared-memory machines. Research Report UCB/CSD 88/408, University of California, Berkeley, 1988.

[McD97] C. McDiarmid. Probability. In L. Beineke and R. Wilson, editors, *Graph Connections*, chapter 13, pages 194–207. Oxford Science Publications, 1997.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[RS85] L. Rudolph and W. Steiger. Subset selection in parallel. In *International Conference on Parallel Processing*, pages 11–14, 1985.