RELATóRIO TéCNICO

A PARALLEL ALGORITHM FOR FINDING SPANNING TREES OF GRAPHS

Jayme luiz Szwarcfiter

NCE-16/90

Junho/90

Universidade Federal do Rio de Janeiro
Núcleo de Computação Eletrônica
Caixa Postal 2324
20001 - Rio de Janeiro - RJ
BRASIL

Este artigo será publicado originalmente no I URSS-BRAZIL Seminar on Informatics, Scientific Computing and Applied Mathematics, que realizar-se-á em Nova Friburgo/RJ, no período de 24 a 27 de julho de 1990.

A PARALLEL ALGORITHM FOR FINDING SPANNING TREES OF GRAPHS

ABSTRACT

We describe a parallel algorithm for finding the connected components of a graph. The algorithm constructs a spanning tree for each of its connected components in $O(\log^2 n)$ time with $O((n+m)/\log n)$ processors and $O(n+m)$ space, under a CREW PRAM model, where n and m are the number of vertices and edges of the graph, respectively.

UM ALGORITMO PARALELO PARA DETERMINAR ARVORES GERADORAS DE GRAFOS

RESUMO

Descrevemos um algoritmo paralelo para determinar os componentes conexos de uma grafo. O algoritmo constroi uma arvore geradora para cada um de seus componentes conexos em tempo $O(\log^2 n)$, com $O((n+m)/\log n)$ processadores e $O(n+m)$ espaco, sob um modelo CREW PRAM, onde n e m representam os numeros de vertices e arestas do grafo, respectivamente.

A PARALLEL ALGORITHM FOR FINDING SPANNING TREES OF GRAPHS

Jayme Luiz Szwarcfiter
Universidade Federal do Rio de Janeiro
Nucleo de Computacao Eletronica / Instituto de Matematica
Caixa Postal 2324, Rio de Janeiro, RJ
CEP 20001 - Brasil

June 1990

ABSTRACT

We describe a parallel algorithm for finding the connected components of a graph. The algorithm constructs a spanning tree for each of its connected components in $O(\log^2 n)$ time with $O((n+m)/\log n)$ processors and $O(n+m)$ space, under a CREW PRAM model, where n and m are the number of vertices and edges of the graph, respectively.

RESUMO

Descrevemos um algoritmo paralelo para determinar os componentes conexos de uma grafo. O algoritmo constroi uma arvore geradora para cada um de seus componentes conexos em tempo $O(\log^2 n)$, com $O((n+m)/\log n)$ processadores e $O(n+m)$ espaco, sob um modelo CREW PRAM, onde n e m representam os numeros de vertices e arestas do grafo, respectivamente.

1. Introduction

Finding connected components of a graph is commonly considered to be the most basic algorithmic graph problem. Many parallel algorithms have been devised for it. The most efficient of them are based on an approach by Hirshberg, Chandra and Sarwate [5] (cf. Karp and Ramachandran [6]). Among those, the algorithm with least complexity was obtained by applying the optimal list ranking techniques by Cole and Vishkin [3]. This provides an implementation of the methods by Awerbuch and Shiloach [2] and Shiloach and Vishkin [7] which runs in time $O(\log n)$ using $O((n+m).\alpha(m,n)/\log n))$ processors, for an arbitrary winner CRCW PRAM model. Here, n and m are the number of vertices and edges of the graph, respectively, while $\alpha(m,n)$ is a very slowly growing function, the inverse Ackerman function. By the simulation among models, this implies an algorithm for the EREW PRAM model of complexity $O(\log^2 n)$ time and $O((n+m).\alpha(m,n)/\log n)$ processors.

In this paper we describe a variation of the common approach for the connectivity problem. It leads to an algorithm which can be implemented in a CREW PRAM model, using $O(\log^2 n)$ time and $O((n+m)/\log n))$ processors. The implementation is based on the optimal techniques for list ranking (Cole and Vishkin [3]), Euler tours on trees (Tarjan and Vishkin [8]) and sorting (Ajtai, Komlos and Szemeredi [1]). Parallel algorithmic techniques were surveyed by Eppstein and Galil [4] and Karp and Ramachandran [6].

Basically, the proposed algorithm constructs iteratively a spanning tree for each connected component of the graph. At each iteration, a convenient forest is selected. The connected components of the forest are then obtained and added to the spanning trees under formation.

2. The Algorithm

G denotes an undirected graph, with vertex set V(G) and edge set E(G), $|V(G)| = n$ and $|E(G)| = m$. A *spanning forest* of G is a subgraph $F \subset G$ such that $V(F) = V(G)$ and each of its connected components is a tree. If for every edge $(v,w) \in E(G)$, v and w lie in the same tree of F then F is *complete*; if they lie in different ones then F is *trivial*. A trivial forest is denoted by F_{ϕ}. Starting from it we describe an algorithm to construct a complete forest.

Any spanning forest F induces a labelling $f: V \to \{1, \ldots, n\}$, as follows: $f(v) = f(w)$ iff v and w belong to a same tree of F. An edge $(v,w) \in E(G)$ is *active* when $f(v) \neq f(w)$. In this case, letting $f(v) < f(w)$, (v,w) is an edge of *type* $(0,f(v))$ and $(1,f(w))$. An *active* vertex is one incident with an active edge. Denote by n'(G) the number of such vertices of G. A *transversal* H of F having *type* p, $0 \le p \le 1$, is a subgraph of G such that

(i) all edges of G are active, and

(ii) each $v \in V(G)$ is incident in H with at most one edge of type $(p,f(v))$.

Figure 1 illustrates the two types of transversals.

The application of transversals in the process of determining a complete spanning forest of G is based on the following theorem.

Theorem 1: Let G be a graph, F a spanning forest of it and H a transversal of F. Then $F \cup H$ is a spanning forest.

Proof: Assume first that H is of type 0. Since F is a spanning subgraph of G, so is $F \cup H$. We have then to show that $F \cup H$ has no cycles. Suppose this is not true, and let C be the cycle of $F \cup H$ of the form $v_1, \ldots, v_q, v_1$, $q > 2$. Let $F' = F \cup C$ and $H' = H \cup C$. Without loss of generality, let v_1 be a vertex of least label in C. Since F is a forest, C must contain an edge of H incident with each distinct label of C. Therefore, again without loss of

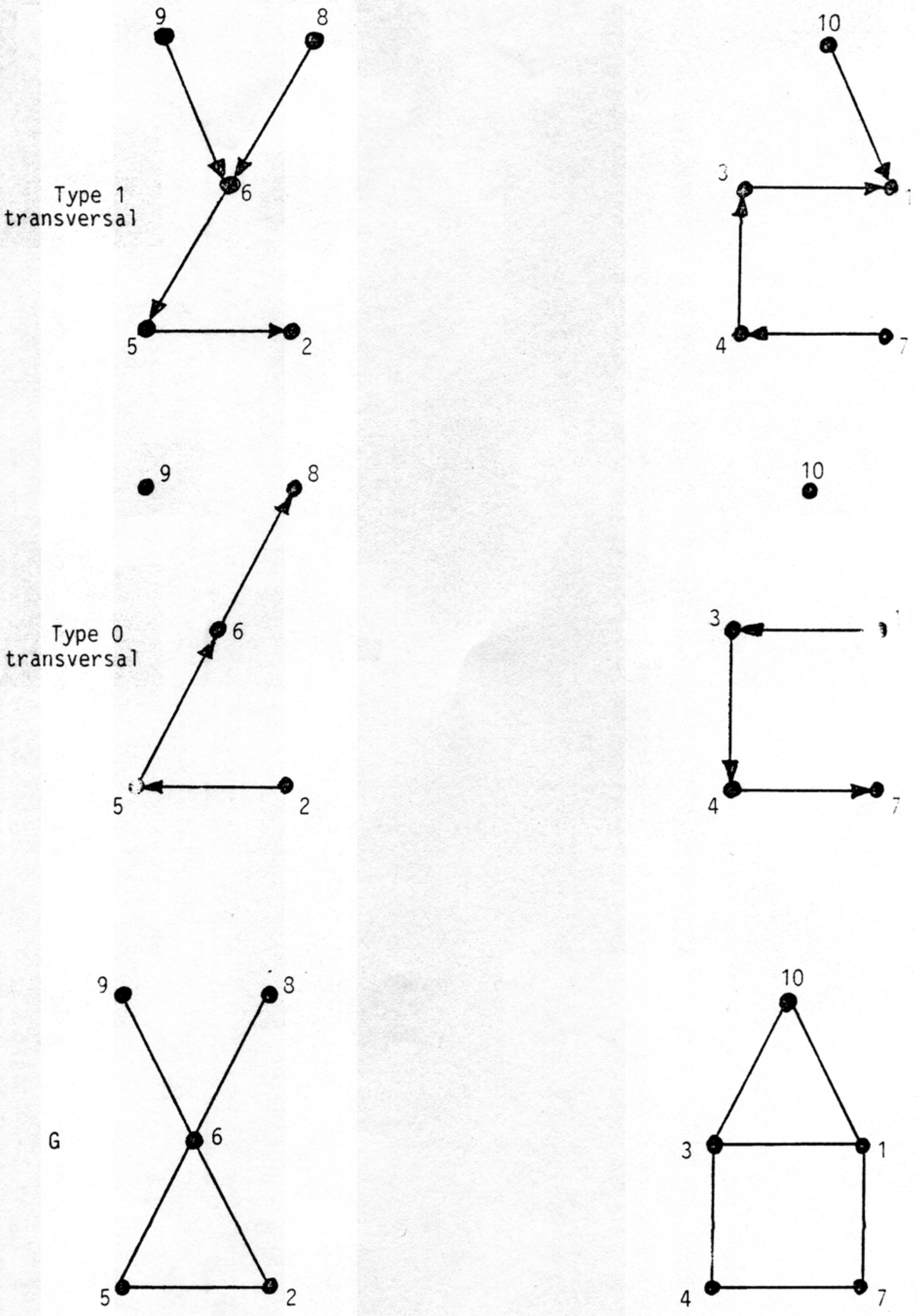

FIGURE 1:
A GRAPH G AND TWO TRANSVERSALS

generality we may assume that $(v_1,v_2) \in E(H')$. Hence $f(v_1) < f(v_2)$, that is (v_1,v_2) is an edge of type $(0,f(v_1))$. Also, $f(v_q) = f(v_1)$, otherwise $f(v_1) < f(v_q)$ which implies $(v_1,v_q) \in E(H')$, i.e. (v_1,v_q) is a second distict edge of H' having type $(0,f(v_1))$. The latter contradicts H as a type 0 transversal. Now, let k be the smallest integer, $2<k\leq q$, such that $f(v_i) = f(v_q)$ for $k\leq i\leq q$ and $f(v_{k-1}) \neq f(v_k)$. Such a value of k exists, since $f(v_2) \neq f(v_1) = f(v_q)$. Hence $f(v_{k-1}) > f(v_k)$. That is, (v_{k-1},v_k) is again a second distinct edge of H' to be of type $(0,f(v_1))$, a contradiction. Therefore, $F \cup H$ is acyclic. If H is of type 1, use similar arguments. $\square$

The above theorem leads to an algorithm for finding a complete spanning forest of a graph G. Start from the trivial forest $F = F_\phi$ and iteratively find a transversal H of F, which is incorporated to F. The process terminates when G has no longer active vertices.

A necessary condition for this algorithm to be efficient is that it has to terminate within a logarithmic number of iterations. Next, we show that this condition is satisfied.

A *maximum transversal* is one with a maximum number of edges.

Theorem 2: Let G be a graph with no isolated edges and H a maximum transversal of F_ϕ. Then $|E(H)| \geq n/2$.

Proof: Suppose that H is of type 0, otherwise the proof is similar. Since F_ϕ is trivial all vertices of G have different labels. We show that $V(G)-V(H)$ is an independent set of G. Assume it is not and let $w_1, w_2 \in V(G)-V(H)$, $f(w_1) < f(w_2)$ and $(w_1,w_2) \in E(G)$. Then (w_1,w_2) is an edge of type $(0,f(w_1))$ and since $w_1 \notin V(H)$ it follows $H \cup \{(w_1,w_2)\}$ is a type 0 transversal with one more edge than H, a contradiction. Hence $V(G)-V(H)$ is an independent set. Now, let $q\geq 1$ be the number of connected components (trees) of H and v_i, $1\leq i\leq q$, be the vertex of maximum label in the i-th component of H. Then $\{v_1,\dots,v_q\}$ is also an independent set. Otherwise, if $(v_i,v_j) \in E(G)$ then (v_i,v_j) is an edge of type $(0, f(v_1))$ and since H does not contain any other edge of such type, it follows that $H \cup \{(v_1, v_2)\}$ is also a type 0 transversal, a contradiction. A similar argument proves that $I = \{v_1, \ldots, v_q\} \cup [V(G) - V(H)]$ is still an independent set. Now, $|I| = q + n - |V(H)|$. Since $|E(H)| = |V(H)| - q$, it follows $|I| = n - |E(H)|$. If $|E(H)| < n/2$ then $|I| \geq n/2$. Since G has no isolated vertices and I is an independent set, each $w_i \in I$ must be adjacent to a vertex $w_j \notin I$. Since H is maximum and of type 0, it follows $f(w_i) > f(w_j)$. Let H' be the subgraph defined by selecting one such (w_i, w_j) edge, for each $w_i \in I$. Then H' is a type 1 transversal of F_ϕ having $|I| > |E(H)|$ edges, a contradiction. Therefore $|E(H)| \geq n/2$. □

Finding a maximum transversal is simple. Construct a maximum transversal H of each type, 0 and 1, and then choose the largest of them. To obtain H, for $1 \leq i \leq n$, select if possible as a representative of v_i, any edge (v_i, v_j) such that $f(v_i) < f(v_j)$ if H is of type 0, or $f(v_i) > f(v_j)$ otherwise.

The *condensation* $G^*(F)$ is the graph obtained from G by identifying all vertices which lie in a same tree of F. It follows that if H is a transversal of some spanning forest F of G then it is also a transversal of the trivial forest of $G^*(F)$. We can now formulate the algorithm.

Let G be a given graph. In the *initial stage* define $F := F_\phi$. In the *general stage* if $n'(G) = 0$ the algorithm terminates (F is a complete forest of G). Otherwise, find a maximum transversal H of the empty forest of G, set $F := F \cup H$, $G := G^*(H)$ and repeat the general stage. □

Theorem 2 asures that the number of iterations of the general stage is at most $1 + \lfloor \log_2(n-1) \rfloor$.

3. Implementation

In this section we describe an implementation of the algorithm in a CREW model.

The graph G is supposed to be given by a set of adjacency lists $A_G(v_i)$, $1 \le i \le n$. Each node $v_j \in A_G(v_i)$ keeps v_j itself, a pointer to the next node of this list and a value $g(v_j)$, possibly equal to the current label of v_j. The output complete forest F is also represented by its adjacency lists $A_F(v_i)$. In addition, the current labels are kept in an n-element vector f.

INITIAL STAGE:

In the initial stage, we set $A_T(v_i) := \phi$ and $f(v_i) := i$, $1 \le i \le n$. Next, we examine the adjacency lists $A_G(v_i)$ and in each node corresponding to $v_j \in A_G(v_i)$ set $g(v_j) := f(v_j)$, $1 \le i \le n$. Finally, compute $n'(G)$ by counting the number of vertices v_i such that $A_G(v_i) \ne \phi$. The operations in this stage require $O(\log n)$ time with $O((n+m)/\log n)$ processors, using list ranking.

GENERAL STAGE:

The general stage is formed by the three steps below.

a. *Finding a Maximum Transversal h:*

The chosen transversal is to be given by a vector h, where $h(v_i)$ is the label of the representative of vertex v_i in H, or $h(v_i) = 0$ when v_i has no representative. We also keep the actual representative of v_i, denoted by $r(v_i)$. The computation of h is as follows.

To each edge defined by $v_j \in A_G(v_i)$ we assign mark 0 if $f(v_i) < g(v_j)$, or mark 1 when $f(v_i) > g(v_j)$. Then we form sublists $A_G^0(v_i)$ consisting of all nodes of $A_G(v_i)$ having mark 0, and sublists $A_G^1(v_i)$ with those marked 1.

Next, for $0 \le q \le 1$ construct two vectors s^0 and s^1:

$$s^q(v_i) = \begin{cases} 1, \text{ if } A_G(v_i) \ne \phi \\ 0, \text{ otherwise,} \end{cases} \quad \text{for } 1 \le i \le n.$$

Compute $\sum s^0(v_i)$ and $\sum s^1(v_i)$. Define $q := 0$ if $\sum s^0 \ge \sum s^1$, and $q := 1$ otherwise. That is, the maximum transversal is of type q. Finally, construct h as follows. For each v_i, $1 \le i \le n$, if $A^q_G(v_i) = \phi$ then v_i has no representative and $h(v_i) := 0$; otherwise set $h(v_i) := g(v_j)$ and $r(v_i) := v_j$, where v_j is the vertex of the first node in $A_G(v_i)$.

Using optimal list ranking, this step can be implemented in $O(\log^2 n)$ time with $O((n+m)/\log n)$ processors, considering all iterations of the general stage.

b. *Computing $F := F \cup H$:*

In this step we identify all vertices of G belonging to a same tree of H and assign them a common label. We find a linear ordering (say, preorder) on the vertices of the trees of H and use it to guide the incorporation of H in T. The operations involved are the following.

First, perform the actual union of the forests. Simply, for $1 \le i \le n$, if $h(v_i) \ne 0$ then include $r(v_i)$ in the adjacency list $A_F(v_i)$.

Next, compute a preorder traversal of the trees of H. For this operation, we are required to construct the adjacency lists $A_H(v_i)$ of H, with each edge (v_i, v_j) represented twice: $v_i \in A_H(v_j)$ and $v_j \in A_H(v_i)$. To obtain the required structure, define a list L having $2.|E(H)|$ nodes, as follows. Each v_i such that $h(v_i) \ne 0$ contributes with two nodes to L, namely those corresponding to the edge $(v_i, h(v_i)) \in E(H)$. The lists are formed after sorting L.

The actual preorder traversal is to be given by a vector p, in the following manner: for $1 \le i \le n$, if $v_i \notin V(H)$ then $p(v_i) := 0$; otherwise $p(v_i)$ contains the preorder successor of v_i in H. All trees of H are dealt with in parallel.

Now we proceed to update the labels . The idea is to choose a representative label of each tree of H (say, the label of its root in the preorder traversal) and assign it to the labels of all vertices of the tree. First, compute a vector *root*, such that if $p(v_i) = 0$ then $root(v_i) := 0$; otherwise assign to $root(v_i)$ the root label of the tree of H which contains v_i. Finally, for each vertex v_i such that $root(v_i) \neq 0$, set $f(v_i) := f(root(v_i))$.

Using appropriate list ranking, Euler tour on trees and sorting, we can implement this step in $O(\log^2 n)$ time with $O(n/\log n)$ processors, overall. Observe that the sum of the sizes of the list L, over all iterations of the general stage, is at most $2(n-1)$. Therefore, the overall number of operations required for sorting all lists L is $O(n \log n)$. This agrees with the above bounds, A similar argument applies to the computation of vector root.

c. *Computing $G := G^*(H)$:*

In this step, we link together the adjacency lists of the vertices of G which now belong to a same tree of T. They correspond to those having a common label. The idea is to perform this operation guided by the preorder traversal of H, given by p. For $1 \le i \le n$, if $p(v_i) = v_j \neq 0$ then we incorporate $A_G(v_j)$ in $A_G(v_i)$, by making the last node of $A_G(v_i)$ to point to the first of $A_G(v_j)$, and redefining $A_G(v_j) := \phi$.

Next, we update the values of $g(v_j)$ as follows. For $1 \le i \le n$, if $A_G(v_i) \neq \phi$ then examine each node $v_j \in A_G(v_i)$. If $root(v_j) \neq 0$ set $g(v_j) := root(v_j)$.

Finally, compute the new value of $n'(G)$.

This step can be implemented in overall time $O(\log^2 n)$ using $O((n+m)/\log n)$ processors, again by optimal list ranking. Hence these are the bounds for the entire algorithm.

4. Conclusions

We have described a parallel algorithm for finding a spanning tree of each connected component of a graph, together with an implementation of complexity $O(\log^2 n)$ time with $O((n+m)/\log n)$ processors and $O(n+m)$ space, under a CREW PRAM model. Clearly, the connected components of the graph are obtained as a by-product of the spanning trees.

REFERENCES

1. M. Ajtai, J. Komlos and E. Szemeredi, Sorting in c log n Parallel Steps, *Combinatorica* 3 (1983), pp. 1-19.

2. B. Awebuch and Y. Shiloach, New Connectivity and MSF Algorithms for Shuffle Exchange Network and PRAM, *IEEE Trans. on Computers* C-36 (1987), pp. 1258-1263.

3. R. Cole and U. Vishkin, Approximate and Exact Parallel Scheduling with Applications to List, Tree and Graph Problems, *Proc. 27th Annual IEEE Symp. on Foundations of Comp. Sci.*, 1986, pp. 478-491.

4. D. Eppstein and Z. Galil, Parallel Algorithmic Techniques for Combinatorial Computation, *Ann. Rev. Comp. Sci.* 1988, pp. 233-283.

5. D. S. Hirshberg, A. K. Chandra and D. V. Sarwate, Computing Connected Components on Parallel Computers, *CACM* 22 (1979), pp. 461-464.

6. R. M. Karp and V. Ramachandran, *Parallel Algorithms for Shared Memory Machines*, Rep. UCB/CSD 88/408, Computer Sc. Div., University of California, Berkeley, CA, 1988.

7. Y. Shiloach and U. Vishkin, An O(log n) Parallel Connectivity Algorithm, *J. Algorithms* 3 (1982), pp. 57-67.

8. R. E. Tarjan and U. Vishkin, An Efficient Parallel Biconnectivity Algorithm, *SIAM J. Comp* 14 (1985), pp. 862-874.