A Coarse-Grained Parallel Algorithm for Spanning Tree and Connected Components*

E. N. Cáceres[1], F. Dehne[2], H. Mongelli[1], S. W. Song[3], and J. L. Szwarcfiter[4]

[1] Universidade Federal de Mato Grosso do Sul, Campo Grande, Brazil,
edson@dct.ufms.br,
http://www.dct.ufms.br/∼edson,
mongelli@dct.ufms.br,
http://www.dct.ufms.br/∼mongelli
[2] Carleton University, Ottawa, Canada K1S 5B6,
frank@dehne.net,
http://www.dehne.net
[3] Universidade de São Paulo, São Paulo, Brazil,
song@ime.usp.br,
http://www.ime.usp.br/∼song
[4] Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brazil,
jayme@nce.ufrj.br,
http://www.cos.ufrj.br/docentes/jayme.html

Abstract. Computing a spanning tree and the connected components of a graph are basic problems in Graph Theory and arise as subproblems in many applications. Dehne et al. present a BSP/CGM algorithm for computing a spanning tree and the connected components of a graph, that requires $O(\log p)$ communication rounds, where p is the number of processors. It requires the solution of the Euler tour problem which in turn is based on the solution of the list ranking problem. In this paper we present a new approach that does not need to solve the Euler tour or the list ranking problem. It still requires $O(\log p)$ communication rounds and has the practical advantage of avoiding the list ranking computation. Rather it is based on the integer sorting algorithm which can be implemented efficiently on the BSP/CGM model.

1 Introduction

Computing a spanning tree and the connected components of a graph are basic problems in Graph Theory and arise as subproblems in many applications. The sequential algorithms use depth-first or breadth-first search to solve these problems efficiently. The parallel solutions for these problems, however, do not use these search methods because they are not easy to parallelize. They are based instead on the approach proposed by Hirschberg et al. [1], where super-vertices

* Partially supported by FINEP-PRONEX-SAI Proc. No. 76.97.1022.00, FAPESP Proc. No. 1997/10982-0, CNPq Proc. No. 30.5218/03-4, 55.2028/02-9, and the Natural Sciences and Engineering Research Council of Canada.

of the graph are successively combined into larger super-vertices. The approach gives rise to algorithms for PRAM models [2]. The most efficient of these algorithms is on a CRCW PRAM of $O(\log n)$ time with $O((m+n)\alpha(m,n))/\log n$ processors, where $\alpha(m,n)$ is the inverse of the Ackermann's function [2].

Dehne et al. [3] present a BSP/CGM algorithm for computing a spanning tree and the connected components of a graph, that requires $O(\log p)$ communication rounds, where p is the number of processors. The algorithm in [3] requires the solution of the Euler tour problem which in turn is based on the solution of the list ranking problem.

In this paper we present a new approach that does not need to solve the Euler tour or the list ranking problem. It still requires $O(\log p)$ communication rounds and has the practical advantage of avoiding the list ranking computation which, in spite of presenting an $O(\log p)$ communication rounds complexity, has been shown to require large constants in practical implementations. The proposed algorithm is based on the integer sorting algorithm which can be implemented efficiently on the BSP/CGM model [4]. We use a special spanning forest, called *strut* [5], of a bipartite graph. The proposed algorithm uses a strut to compute a spanning tree and the connected components of a given graph. We first assume a bipartite graph as input. Then we show how to handle the case of a general graph by transforming it into a corresponding bipartite graph. If the obtained bipartite graph is not connected, the algorithm will compute a spanning tree for each connected component.

In the next sections we present the parallel computing model, the definitions used, and the main result of this paper.

2 Coarse-Grained Multicomputer (CGM) Model

We consider a version of the BSP model [6], called the *Coarse-Grained Multicomputer* (CGM) model [7]. It uses two parameters: the input size N and the number of processors p. In comparison to the BSP model, the CGM allows only bulk messages in order to minimize message overhead.

A CGM consists of a set of p processors each with its local memory. Each processor is connected by a router that can send messages in a point-to-point fashion (or shared memory). A CGM algorithm consists of alternating local computation and global communication rounds separated by a barrier synchronization. In a computing round, we usually use the best sequential algorithm in each processor to process its data locally. In each communication round each processor sends $O(N/p)$ data and receives $O(N/p)$ data. Therefore, in terms of the BSP terminology, each communication round consists of routing a single h-relation with $h = O(N/p)$. We require that all information sent from a given processor to another processor in one communication round be packed into one long message, thereby minimizing the message overhead. In the CGM model, the communication cost is modeled by the number of communication rounds.

3 Some Definitions and the Main Idea

Consider a bipartite graph $H = (V_1, V_2, E)$ with vertex sets V_1 and V_2 and edge set E where each edge joins one vertex of V_1 and one vertex in V_2. If v is a vertex of a subgraph H' of H, then $d_{H'}(v)$ denotes the degree of v in H'. Let the vertices of V_1 be $u_1, u_2, \cdots, u_{n_1}$ and the vertices of V_2 be $v_1, v_2, \cdots, v_{n_2}$.

We define a *strut ST in* V_1 as a spanning forest of H such that each $v_i \in V_2$ is incident in ST with exactly one edge of E, and (u_j, v_i) is an edge of ST implies (u_k, v_i) is not an edge of H, for any $u_k \in V_1$, $k < j$.

To define a *strut in* V_2, the roles for the sets V_1 and V_2 in the above definition are exchanged.

A vertex $u \in V_1$ is called *zero-difference* in ST if $d_H(u) - d_{ST}(u) = 0$. Otherwise, the vertex is referred to as *non-zero-difference*.

Before we present the CGM algorithm, let us give some ideas and give a simple example.

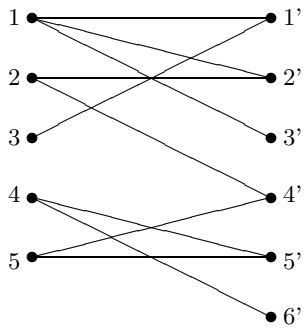

Fig. 1. Bipartite graph $H = (V_1, V_2, E)$

Consider the input bipartite graph $H = (V_1, V_2, E)$ of Fig. 1. We have
$V_1 = \{1, 2, 3, 4, 5\}$, $V_2 = \{1', 2', 3', 4', 5', 6'\}$ and
$E = \{(1, 1')(1, 2')(1, 3')(2, 2')(2, 4')(3, 1')(4, 5')(4, 6')(5, 4')(5, 5')\}$.

We can first compute a spanning forest for $H = (V_1, V_2, E)$, obtained by determining a strut ST in H. We obtain a strut (see Fig. 2), which is represented by solid lines while the remaining edges are represented by dash lines.

Now compute the zero-difference vertices in V_1. Consider vertex 1. All the (three) edges in H incident with this vertex is also in ST. Thus $d_H(1) - d_{ST}(1) = 0$ and vertex 1 is zero-difference. Likewise vertex 4 is also zero-difference. Vertex 2 has $d_H(2) - d_{ST}(2) = 2 - 1 = 1$ and thus is not zero-difference.

In the example we have two zero-difference vertices. If, however, we have only *one* zero-difference vertex, then the problem is easily solved by adding to ST one arbitrary edge of $H - ST$ incident to each non-zero-difference vertex of ST.

In case there are two or more zero-difference vertices we can do the following. For each zero-difference vertex $u \in V_1$ compact all the vertices $v_i \in V_2$ incident with u by compressing all the vertices v_i onto the smallest of the v_i.

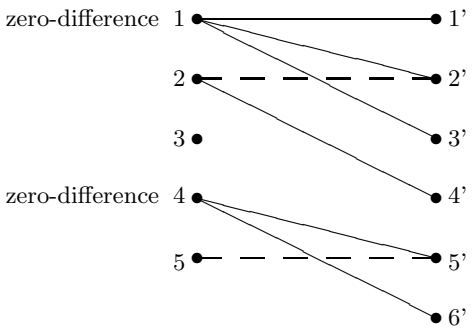

Fig. 2. A strut ST represented by the solid lines.

This can be done repeatedly until only one zero-difference vertex remains.

4 The CGM Algorithm for Bipartite Graphs

Consider a bipartite graph $H(V_1, V_2, E)$ where $|V_1| = n_1$, $|V_2| = n_2$ and $|E| = m$. Consider p processors with local memory of size $O(m/p)$ or $O((n_1 + n_2)/p)$. We present a CGM algorithm (see Algorithm 1) to compute a spanning tree of a bipartite graph. It takes $O(\log p)$ communication rounds and is based essentially on an integer sorting algorithm. Thus it differs from a previous algorithm [3] that requires Euler tour and list ranking. Though the presented algorithm still presents the same complexity on the number of communication rounds, we expect better experimental results since it depends on the sorting algorithm that can be implemented efficiently on the CGM model.

It is instructive to consider the example of Fig. 1 to illustrate the several steps of Algorithm 1.

As storage structures, consider the vectors $EDGE$ and $EDGE'$, each containing m elements. Each element is an edge represented by (u, v), with $u \in V_1$ and $v \in V_2$. The edges of E are stored in vector $EDGE$. In our example, the vector $EDGE$ is

$$(1,1')(1,2')(1,3')(2,2')(2,4')(3,1')(4,5')(4,6')(5,4')(5,5')$$

Initially we make a copy of $EDGE$ in $EDGE'$.

Consider first Phase I.

Lines 3 to 6 of Algorithm 1 obtain a strut ST. First we sort the edges in $EDGE'$ lexicographically in the following way. Given two edges (i, j) and (k, l) then $(i, j) < (k, l)$ when $j < l$ or $((j = l)$ and $(i < k))$. This is done in line 3 of Algorithm 1. Vector $EDGE'$ contains the sorted edges:

$$(1,1')(3,1')(1,2')(2,2')(1,3')(2,4')(5,4')(4,5')(5,5')(4,6')$$

Lines 4 to 6 find a strut ST in V_1. It is represented by the *marked* edges (solid lines of Fig. 2). Vector $EDGE'$ represents ST and for our example, we have:

Algorithm 1 CGM Algorithm for Spanning Tree

Input: A bipartite graph $H(V_1, V_2, E)$ where $V_1 = \{u_1, \ldots, u_{n_1}\}$, $V_2 = \{v_1, \ldots, v_{n_2}\}$ and $|E| = m$. An edge (u_i, v_i) of E has a vertex u_i in V_1 and a vertex v_i in V_2. The m edges are equally distributed among the p processors at random.
Output: A spanning tree of G.
Phase I:

1: Initialize $\bar{V_1} := V_1$ and $\bar{V_2} := V_2$ and $\bar{E} := E$.
2: **for** $\log p$ times **do**
3: Sort the edges (u, v) of $\bar{E}$ by v and then by u.
4: **for** each v_i of V_2 **do**
5: Choose the smallest vertex u_j among all edges (u, v_i) and mark the edge (u_j, v_i). Let ST be the set of the marked edges.
6: **end for**
7: Compute the degree of each vertex $u \in \bar{V_1}$ in $H(\bar{V_1}, \bar{V_2}, \bar{E})$.
8: Compute the degree of each vertex $u \in \bar{V_1}$ in $H_{ST}(\bar{V_1}, \bar{V_2}, ST)$.
9: Using the degrees computed in the previous steps compute the number of zero-difference vertices.
10: **if** number of zero-difference vertices = 1 **then**
11: the algorithm finishes
12: **end if**
13: Compact the graph to produce the compacted graph $H(\bar{V_1}, \bar{V_2}, \bar{E})$.
14: **end for**

Phase II:

1: Compute a spanning forest with the edges that do not belong to ST and removing those with degree($\bar{u}$)=1 where $\bar{u} \in \bar{V_1}$.
2: **for** i:=0 to $\log p$ **do**
3: Processor $2i + 1$ sends its spanning forest to Processor $2i$.
4: Processor $2i$ computes a new spanning forest.
5: **end for**

$$(1,1')(1,2')(1,3')(2,4')(4,5')(4,6')$$

A strut ST in V_1 determines a spanning forest of H.

Lines 7 to 9 find the zero-difference and non-zero-difference vertices of the strut ST. We determine the degrees of each of the vertices in V_1 and store in D_H. In our example $D_H = (3, 2, 1, 2, 2)$.

Determine now which vertices of V_1 are zero-difference. For this, determine the degree of each of the vertices of V_1 in $EDGE'$ and store in D_{ST}. Again for our example, $D_{ST} = (3, 1, 0, 2, 0)$.

Thus the zero-difference vertices are vertices $\{1, 4\}$ and the non-zero-difference vertices are vertices $\{2, 3, 5\}$.

Line 13 produces a compacted graph. For each zero-difference vertex $u \in V_1$ compact all the vertices $v_i \in V_2$ incident with u by merging all the vertices v_i onto the smallest of the v_i. The new compacted graph $H(\bar{V}_1, \bar{V}_2, \bar{E})$ is shown in Fig. 3. It is instructive to note that vertices $2'$ and $3'$ are compressed onto vertex $1'$ and therefore the original edge $(2, 2')$ now becomes $(2, 1')$.

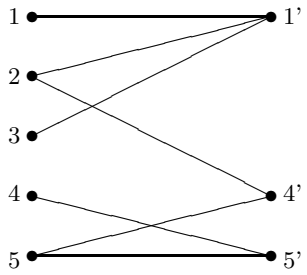

Fig. 3. The compacted graph $H(\bar{V}_1, \bar{V}_2, \bar{E})$.

5 Spanning Trees for General Graphs and Connected Components Algorithms

As mentioned earlier we can transform any graph into a bipartite graph. Consider a non-bipartite graph. We can subdivide each of the edges of the graph by adding a new vertex on each edge. If we consider the vertices of the original graph as belonging to set V_1 and the new added vertices as V_2, then we have a resulting bipartite graph. Thus given a graph $G = (V_1, E)$, substitute the edges $(i, j) \in E$ by two edges (i, k) and (k, j) and consider vertex $k \in V_2$. The graph obtained $H = (V_1, V_2, E')$ is bipartite. Thus we can apply the proposed algorithm. Fig. 4 presents an example with the original graph $G = (V_1, E)$ and the bipartite graph $H = (V_1, V_2, E')$, where $V_1 = \{1, 2, 3, 4, 5\}$ and $V_2 = \{\bar{1}, \bar{2}, \bar{3}, \bar{4}, \bar{5}, \bar{6}, \bar{7}\}$.

The proposed algorithm for computing a spanning tree can be used to determine the connected components of a graph.

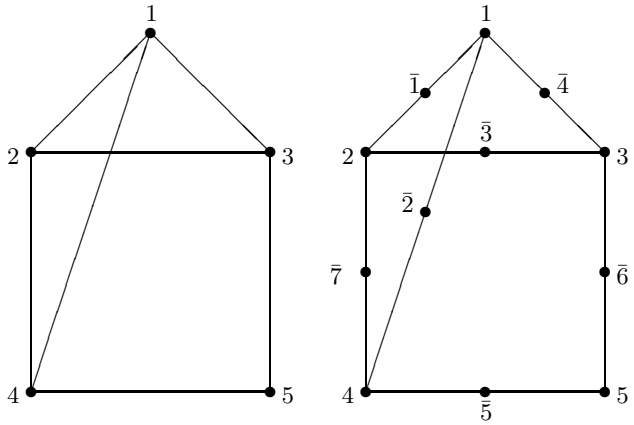

Fig. 4. Original graph $G = (V, E)$ and bipartite graph $H = (V_1, V_2, E')$

In each iteration, the spanning tree algorithm determines each of the sublists of $EDGE'$ formed by edges $(u, v), u = u_i$ that forms a tree, labeled by $EDGE'_{u_i}$. At the end of the algorithm, we can represent each tree with the smallest vertex. Each of the different vertices represent a connected component of the graph.

6 Discussion of the Algorithm

Lemma 1. *Let $H = (V_1, V_2, E)$ be a bipartite graph and let S be a strut in V_1. Let H' be the graph obtained from H by adding to S exactly one edge from $E - S$ that is incident to each non zero-difference vertex of V_1. Then H' is acyclic. Moreover, if V_1 contains exactly one zero-difference vertex then H' is a spanning tree of H.*

Proof.

We can see that S is a set of stars whose centers are vertices in V_1. When we add exactly one edge from $E - S$ incident to each non zero-difference vertex of V_1, the star whose center v_j is a non-zero-difference vertex will be connected to at most a different star of center v_i. Then, the resultant graph H' is acyclic.

By the definition of strut, the vertex degree of each vertex in V_2 is exactly one, so there will be exactly $|V_2|$ edges in S. As V_1 has exactly one zero-difference vertex, the number of edges that can be added is $|V_1| - |V_2| - 1$ and H' is a spanning tree of H. □

Theorem 1. *The number of zero-difference vertices in $\bar{V_1}$ after step 13 is at least divided by 2 in each iteration.*

Proof.

Let $\bar{V_1}'$ and $\bar{V_2}'$ be the partitions before step 13 and let $\bar{V_1}$ and $\bar{V_2}$ be the partitions after step 13. Let k be the number of zero-difference vertices in V_1.

After step 13, the number of vertices in $\bar{V_2}$ is also k since after compaction, for each zero-difference vertex in $\bar{V_1}$, only the vertex in $\bar{V_2}'$ with smallest label is kept in $\bar{V_2}$.

Since all the vertices in $\bar{V_1}$ have degree greater than 1, the number of zero-difference vertices in $\bar{V_1}$ is $|\bar{V_2}|$, e.g., $k/2$. □

Theorem 2. *The Algorithm 1 computes the spanning tree of* $H = (V_1, V_2, E)$ *with* $O(\log p)$ *communication rounds and* $O((m+n)/p)$ *local memory.*

Proof. Phase I of the algorithm can be executed in $O(\log p)$ communication rounds by using a sorting algorithm that takes constant number of communication rounds (e.g. [4]) in each iteration.

The m input edges are distributed equally among the p processors. Each processor thus contains edges (u, v) where $u, v \in V_1 \cup V_2$. The size of $V_1 \cup V_2$ is $n_1 + n_2$ which can be larger than the local memory of each individual processor. It can be shown that after $\log p$ iterations, the number of remaining vertices of $\bar{V_1} \cup \bar{V_2}$ is bounded by $O((n_1 + n_2)/p)$. Thus at the end of phase I, since each processor contains edges (u, v) where $u, v \in \bar{V_1} \cup \bar{V_2}$, all the remaining vertices are stored in every processor. At this time we proceed to phase II.

Phase II consists of $O(\log p)$ communication round during which the individual spanning trees in the processors are combined. □

7 Conclusion

We have presented a CGM algorithm for computing a spanning tree of a graph and its connected components. It takes $O(\log p)$ communication rounds and is based on an integer sorting algorithm. Thus it differs from a previous algorithm [3] that requires the computation of Euler tour and list ranking. Though the presented algorithm still presents the same complexity on the number of communication rounds, we expect it to give good experimental results since it depends on the sorting algorithm that has efficient CGM implementations.

References

1. Hirschberg, D.S., Chandra, A.K., Sarwate, D.V.: Computing connected components on parallel computers. Comm. ACM **22** (1979) 461–464
2. Karp, R.M., Ramachandran, V.: 17. In: Handbook of Theoretical Computer Science - J. van Leeuwen (ed.). Volume A. Elsevier/MIT Press (1990) 869–941
3. Dehne, F., Ferreira, A., Cceres, E., Song, S.W., Roncato, A.: Efficient parallel graph algorithms for coarse grained multicomputers and bsp. Algorithmica **33** (2002) 183–200
4. Chan, A., Dehne, F.: A note on coarse grained parallel integer sorting. Parallel Processing Letters **9** (1999) 533–538
5. Cáceres, E.N., Deo, N., Sastry, S., Szwarcfiter, J.L.: On finding euler tours in parallel. Parallel Processing Letters **3** (1993) 223–231
6. Valiant, L.: A bridging model for parallel computation. Communication of the ACM **33** (1990) 103–111
7. Dehne, F., Fabri, A., Rau-Chaplin, A.: Scalable parallel geometric algorithms for coarse grained multicomputers. In: Proc. ACM 9th Annual Computational Geometry. (1993) 298–307