# Another PRAM Algorithm for Finding Connected Components of Sparse Graphs

Marco Bellia and M.Eugenia Occhiuto
Dipartimento di Informatica,
Università di Pisa
e-mail: occhiuto@di.unipi.it

November 20 1997

**Abstract**

We present an algorithm which exploits a new approach to the problem of finding the connected components of an undirected graph, CCug for short, with $v$ vertices and $e$ edges. The algorithm has depth $O(log^2(e))$[1] on a CREW PRAM using $e$ processors, hence its cost is not affected by the number $v$ of graph vertices. This makes the algorithm the one with best speedup and best cost for CCug on highly sparse graphs. On dense graphs conversely, its performance is comparable to the one of the algorithm in [12] and a little worse than the one in [5]. A variant of the algorithm with the same bound but running on the EREW model is also included. The algorithm can be used to find the transitive closure of binary, symmetric relations. In this case $e$ is the number of axioms and $v$ is the range of the relation.

## 1 Introduction

The problem of finding the connected components of undirected graphs is considered one of the most basic problems on graphs [9, 16]. Yet, it occurs as a basic operation in a great variety of applications [21]. In algebra, CCug is strongly related to the fundamental operation of finding the transitive closure of binary, symmetric, (finite) relations. The edges of the underlying graph are the axioms of the relation, the vertices are the points on which the relation is defined, and the connected components of the graph are the

[1] in this paper we denote $Log_2(n)$ by $log(n)$

classes, i.e. disjoint sets of points, of the partition induced by the relation. Algorithms for CCug thus apply in a natural way to computing the transitive closure of binary relations [10, 11, 18].

An important aspect of the extended use of CCug in various applications is that the graphs to which it applies, may have considerable differences in the density of the structure, i.e. the ratio of edges to vertices [22]. Note that the size of a graph, with $v$ vertices and $e$ edges, is $v+e$, and (for undirected, unlabelled graphs) $v^2$ is a superior bound for $e$. In many applications, we can safely assume that CCug deals with graphs which have $e > v - 2$. Nevertheless, in finding the transitive closure of binary relations, there may be very few axioms compared to the points of the relation. This means that the underlined graphs are non dense, i.e. the graphs representing such relations have $0 \leq \lim_{e+v\rightarrow\infty} e/v < k$ for a natural $k$. This shows that the number $v$ of vertices can be a dominant cost factor, namely when graphs are sparse, i.e. the above limit is 0. Unfortunately, the cost of all the PRAM algorithms for CCug, in the literature, is affected by the number $v$ of graph vertices. This means that , at present, algorithms for CCug have a performance which may be good for dense graphs but has a downfall on sparse graphs.

We present a new, deterministic, CREW PRAM algorithm which solves CCug in $O(log^2(e))$ time using $e$ processors. One of its main characteristic is that neither the time nor the number of processors used is affected by the number of vertices of the graph. Our algorithm has a suboptimal, low, TS (Time x Space product) which is affected only by the number of edges of the graph. Since $e \leq v^2$, the algorithm is very competitive for CCug on dense graphs and is the one with best cost for CCug on sparse graphs. The algorithm exploits a new approach based on edge replacement which preserves connected component on the one hand and reduces *multiple edges* [5] on the other hand.

An overview of the approach used in the algorithm is given in Section 3. Sections 4, 5 and 6 are devoted to correctness, soundness, and complexity, respectively. Section 7 treats extensively the computation of the binary tree search used in the algorithm to select edges. A variant of the algorithm is then discussed in Section 8. It has the same bound as the algorithm presented in Section 3 but it is defined for the EREW PRAM model. This fact has several significant aspects which are discussed in Section 9 where a comparative study of the cost of our algorithm is given together with an overview of the algorithms for PRAM currently proposed by [14, 26, 4, 24, 27, 22, 1, 12, 7, 17, 20, 25, 5].

## 2 Preliminaries

A graph is a pair $(\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of vertices and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges. Let $v = |\mathcal{V}|$, i.e. cardinality of $\mathcal{V}$, and $e = |\mathcal{E}|$. Then the size of a graph is $e + v$. Given an edge $a = (x, z)$, vertex $x$ is the source and vertex $z$ is the target of $a$ [2]. A connected component of a graph $G$ is a maximal subgraph $H$ of $G$, i.e. $H = (\mathcal{V}_H \subseteq \mathcal{V}, \mathcal{E}_H \subseteq \mathcal{E})$, for which there exists a path between each pair of vertices in the subgraph. A path is a sequence of adjacent edges, i.e. any sequence $(x_1, z_1), ..., (x_p, z_p)$ of edges such that for each $1 \leq i < p$, $z_i = x_{i+1}$. The size of a path is the number of edges of the sequence. For undirected graphs, adjacency means that edges share a vertex, i.e. $z_i \in \{x_{i+1}, z_{i+1}\}$. An undirected graph always has denumerable infinite different paths between any pair of connected vertices. In considering connectivity, we are only interested in *acyclic paths*, hence hereafter we will refer to paths as meaning a sequence of adjacent, not repeated, edges. Since each vertex belongs to one and only one connected component of the graph, connected components define a partition $\mathcal{P}$ on the set $\mathcal{V}$ of vertices. Hence, for each $x, z \in \mathcal{V}$, $x$ and $z$ belong to the same connected component if and only if $x$ and $z$ belong to the same part of $\mathcal{P}$, i.e. $\mathcal{P}(x) = \mathcal{P}(z)$. We can map $\mathcal{P}$ into $\mathcal{V}$ so that $\mathcal{P}(x) = min\{y \in \mathcal{V} \mid \mathcal{P}(y) = \mathcal{P}(x)\}$ [3]. Our algorithm uses this kind of partition for the output. For the input, different representations of graphs are in common use [15]. Our algorithm uses the most simple and economical pair $(\mathcal{V}, \mathcal{E})$. Moreover, since the algorithm does not make use of V, the input is restricted to one array $E$ which contains only E.

## 3 Algorithm

Given the graph $G$=$(\mathcal{V}, \mathcal{E})$, the algorithm finds the partition $\mathcal{P}$. The input is represented by the $[0, e-1]$ indexed array $E$ of edges, i.e. pairs (*source*,*target*). The output is represented by the $[1, v]$ indexed array $P$ of vertices[4]. The algorithm consists of $\lceil log(e) \rceil$ reductions. Each *reduction*:

[2] We use ordered pairs (x, z) to denote both directed and undirected edges. In the latter case, any undirected edge (x, z) is equivalent, under connectivity, to the undirected (z, x), but differs from it in terms of the source and the target vertices. The distinction between source and target vertex is merely syntactic in undirected edges but it is useful in describing our work.

[3] $min$, $max$ compute minimum and maximum of sets, respectively. Hereafter we assume that vertices are injectively mapped onto the integer interval $[1, v]$

[4] We assume that $P$, due to the effect of memory allocation, is initially set to 0. Thus, for each vertex $x$, $P[x] = 0$ means that $x$ is identically mapped under $P$. This avoids

*makes* a (syntactic) re-orientation of edges so that the source is always greater than the target;

*selects* for each vertex $x$ occurring as a source in $E$, the set $\mathcal{I}(x)$ of indices of all edges with $x$ as their source, i.e. $\mathcal{I}(x)=\{i \mid E[i] = (y,z) \wedge x = y\}$;

*applies* to each $\mathcal{I}(x)$ a $log(e)$ depth binary tree search in which edges may be replaced. At step k, for k$\in [0..\lceil log(e) \rceil - 1]$, it considers all disjoint pairs of contiguous indices $i, j$ in $\mathcal{I}(x)$ whose difference is less than or equal to $2^k$, i.e. $\{i < j \mid i, j \in \mathcal{I}(x)$ and, for $h \in [1, 2^k], i + h \in \mathcal{I}(x)$ iff $j = i + h\}$. Let $E[i] = (x, z)$, $E[j] = (x, w)$ and $m = min\{z, w\}$, $M = max\{z, w\}$. Then, the step

1. replaces $(x, z)$ with $(x, m)$, i.e. $E[i] \leftarrow (x, m)$, and replaces $(x, w)$ with $(M, m)$, i.e. $E[j] \leftarrow (M, m)$ and
2. removes index $j$ from $\mathcal{I}(x)$.

*sets* for each $\mathcal{I}(x)$, $P[x] = m$. Where $m$ is such that $E[i] = (x, m)$ for the index $i$ which *survived* in $\mathcal{I}(x)$ after the removal of the previous step. Moreover, the step removes chains from $P$ by $\lceil log(e) \rceil$ jumpings $P[x] \leftarrow P[P[x]]$ (for all $x$ not identically mapped by $P$)

*replaces* $E[i] \leftarrow (P[x], P[z])$ for all $i$, where $E[i] = (x, z)$.

Figure 2 shows the various steps of the algorithm which is applied to the graph in Figure 1. Figure 2a is the vector $P$ of the partition currently computed on the vertices of $G$. Initially, each vertex is a connected component. Figure 2b is the vector $E$ of the edges of $G$ (syntactically) oriented by *makes*. The allocation of the edges in $E$ is not essential and hence, arbitrarily chosen. Figure 2c contains the sets $\mathcal{I}$ computed by *selects*. Figure 2d shows the graph $G$ with the edges updated by the computation of *applies*. Figure 2c contains the partition $P$ resulting at the end of *sets*. Note that $P$ contains only one connected component whose minimum vertex is 1. In this case only one iteration of *reduction* would be enough to compute the CCug of $G$. Figure 2f contains the edges computed by *replaces* which are all internal edges of the connected component with minimum 1.

---

initialising $P[x] = x$ which would require $v$ processors.

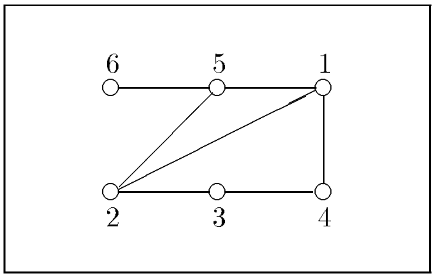


Figure 1: An undirected graph $G$

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $P$ | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 2a: The initial vector $P$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $E$ | 5 | 5 | 4 | 4 | 3 | 2 | 6 |
| | 2 | 1 | 1 | 3 | 2 | 1 | 5 |

Figure 2b: The vector $E$ after the reorientation computed by *makes*

$$
\begin{aligned}
\mathrm{I}(5) &= \{0,1\}\\
\mathrm{I}(4) &= \{2,3\}\\
\mathrm{I}(3) &= \{4\}\\
\mathrm{I}(2) &= \{5\}\\
\mathrm{I}(6) &= \{6\}
\end{aligned}
$$

Figure 2c: The sets $\mathcal{I}$ computed by *selects*

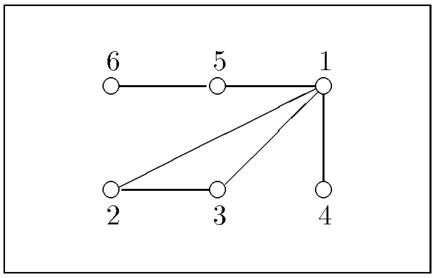


Figure 2d: The Graph $G$ after the updating of $E$ computed by *applies*

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| $P$ | 0 | 1 | 1 | 1 | 1 | 1 |

Figure 2e: The vector $P$ after the computation of *sets*

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $E$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Figure 2f: The vector $E$ after the computation of *replaces*

**Program** CONNECT

**input:** E: an $e \times 2$-entry array which contains the set of edges of an undirected graph, $G$, with $e$ edges and $v$ vertices

**output:** P: an $v$-entry array which contains the connected components to which each vertex belongs

*Notes*: Flag is an $e$-entry array on [0..1] and is used to mark survived edges in E. Vertices are denoted by the naturals in $[1..v]$. P is an array for partition $\mathcal{P}$. Initially, P[i]=0 for all i: value 0 is used as a special tag denoting identity on $\mathcal{P}$, i.e. P[i]=0 means $\mathcal{P}$[i]=i. A conversion operator, $cnv$, is then introduced so that $cnv$(P,$i$)=$\mathcal{P}[i]$. Each connected component of $G$ is expressed by the minimum vertex belonging to it.

```
begin
 For Outer:= 1 to ⌈log(e)⌉ do
   begin
1.   Foreach i ∈ [1..e] Pardo Flag[i]:=1;
2.   Foreach i ∈ [1..e] Pardo
       if E[i,1] < E[i,2] then swap(E[i,1], E[i,2]);
3.   SELECTS&APPLIES (E,Flag);
4.   Foreach i ∈ [1..e] Pardo
       if Flag[i] = 1 then P[E[i,1]]:=E[i,2];
5.   For Inner:= 1 to ⌈log(e)⌉ do
       Foreach i ∈ [1..e] Pardo
         if (P[E[i,1]] ≠ 0) and (P[P[E[i,1]]]≠ 0)
         then P[E[i,1]]:=P[P[E[i,1]]]
6.   Foreach i ∈ [1..e] Pardo
       begin
         E[i,1]:=cnv(P,E[i,1]);
         E[i,2]:=cnv(P,E[i,2]);
       end
   end
end.
```

Figure 3: An $O(log^2(e))$ CREW program for finding connected components with $e$ processors

## 4 On the Soundness

Let $E$ be the input array of the algorithm. We can show that each operation in *reduction* preserves graph connectivity (but not edges).

*re-orientation* is correct since edges are undirected.

*applies* is correct since the sets of edges $\{(x,z),(x,w)\}$ and $\{(x,z),(z,w)\}$ have the same transitive closure, so that edge $(x,w)$ can be replaced in any path by the sequence $(x,z),(z,w)$ (and analogously, since edges are undirected, edge $(z,w)$ can be replaced by the sequence $(x,z),(x,w)$).

*sets* is correct since both the minimum $m$ and any vertex connected to $m$ are in the connected component to which $x$ belongs. (Likewise, the correctness of *replaces* which applies transitivity).

Hence we have the following:

**Lemma 1** *Let $E^1$ be the array of edges resulting from applying* reduction *to E once. Then for each $x \neq z$, a path from $x$ to $z$ exists in $E^1$ if and only if the same holds in $E$.*

## 5 On the Completeness

Let $E$ be the input array of the algorithm for a graph $G$. Let $E^0 \equiv E$, $E^1, ..., E^k$, for $k = \lceil log(e) \rceil$ be the sequence of arrays such that each $E^{i+1}$ is the array resulting from the application of one step of *reduction* to $E^i$. Let $x$ be any vertex of $G$ and let $min_x$ be the minimum vertex of the connected component of $G$ to which $x$ belongs. Then we define $distance(x, E^i)$ to be the size of the maximal path which connects, using only edges in $E_i$, vertex $x$ to $min_x$. Moreover, we define the distance of an array $E^i$ as $distance(E^i) = max\{distance(x, E^i)\}$.

**Lemma 2** *Let $E^{i+1}$ be an array as above. Then*

$$distance(E^i) \geq 2 * distance(E^{i+1}).$$

**Proof** *applies* replaces pairs $(x,z)$, $(x,z_1)$ with $(x,z)$, $(z_1,z)$ (assumed $z$=$min\{z,z_1\}$). Moreover, *replaces* removes all descending chains in $E_i$ (i.e. paths of the form $(x_1,z_1)$, $(z_1,z_2)$, $\ldots$, $(z_{p-1},z_p)$).

**Lemma 3** *Let $E^{i+1}$ be an array as above, and $P^{i+1}$ be the companion partition of $E^{i+1}$ (i.e. the partition computed by step* sets *of the algorithm). Then for each pair of vertices $x \neq z$, $E^i$ contains a path from $x$ to $z$ if and only if $E^{i+1}$ contains a path $(x_1, z_1), ..., (x_p, z_p)$ such that $P^{i+1}[x] \in \{x_1, z_1\}$ and $P^{i+1}[z] \in \{x_p, z_p\}$.*

**Proof** Part *if* follows from lemma1 on correctness. For the part *only if* note that, by definition of *reduction*, if $(y_1, w_1), \ldots, (y_q, w_q)$ is a path of $E^i$ then $(P^i[y_1], P^i[w_1]), \ldots, (P^i[y_q], P^i[w_q)])$ is a path of $E^{i+1}$.

.

**Theorem 1** *Let $P^* = P^{\lceil log(e) \rceil}$ be the partition resulting from the algorithm when we apply it to the array $E$ of the edges of a graph $G$. Then, for each pair of vertices $x \neq z$, $P^*[x] = P^*[z](\neq 0)$ if and only if $x$ and $z$ belong to the same connected component of $G$.*

**Proof** $E$ cannot contain an (acyclic) path longer than $e = |E|$. By lemma 2, $\lceil log(e) \rceil$ iterations of *reduction* traverse all paths in $E$. Moreover, $E$ contains all and only edges of $G$. Hence by lemma 3 we have the theorem.

## 6 On the Complexity

Figure 3 shows a version of the algorithm, where steps 1 to 6 compute *reduction* and are iterated $\lceil log(e) \rceil$ times. In particular, step 2 is *makes*. Steps 4 and 5 are *sets*, and step 6 is *replaces*. In program CONNECT, we introduced the auxiliary array Flag. Its purpose is to mark the indices of $E$ with *survived*, 1, and *not-survived*, 0. Before step 3, all positions are marked survived. Step 3 is *selects* and *applies*: it updates $E$ and resets, in Flag, the entries of all the edges that do not survive, to not-survived. The complexity of the algorithms is discussed for the CREW PRAM model as follows:

- Steps 1, 2, 4 and 6 can be computed in $O(e/p)$ PRAM-Time using $p \leq e$ processors.
- Step 5 can be computed in $O(e * log(e)/p)$ PRAM-Time using $p \leq e$ processors.

The algorithm can thus be computed in $O(e * log^2(e)/p)$ PRAM-Time using $p \leq e$ processors if the complexity of step 3 is not worse than that of step 5.

## 7 On the Computation of SELECTS&APPLIES

Figure 4 shows a program which computes SELECTS&APPLIES. It uses an auxiliary $e \times v$ entry array TEMP. The use of TEMP draws on the *three-way trade-off* technique introduced by [13]. TEMP supplies each source (vertex) of the edges in $E$ with a coloumn of $e$ locations. The entries of the edges sharing the same source, i.e. set $\mathcal{I}$, are collected in the column of TEMP associated with such a source. The computation of *selects* is thus accomplished. Once the edges at the entries which are a multiple of $2^B$, for $B = 0, ..., \lceil log(e) \rceil$, have been selected, we consider the $2^B$ entries of the neighbourhood in the associated column of TEMP. Since neighbourhoods are disjoints, if an edge, say $a_2$, is found in the neighbourhood of a selected edge, say $a_1$, then: *i*) $a_1$ and $a_2$ have the same source; *ii*) $a_2$ cannot be one of the selected edges; *iii*) $a_2$ cannot be in the neighbourhood of another selected edge. Hence *applies* is computed by replacing one of the two edges and resetting the corresponding Flag entry to not-survived. In this way a binary tree search is accomplished, so when $B = \lceil log(e) \rceil$ all the $e$ entries of each column are considered and *applies* is completed. An analytic description of the algorithm is given below.

**Step 1** computes *selects*. It is accomplished by updating only those columns of TEMP which correspond to vertices occurring as sources in $E$. In addition, it sets to i-1 the i-th entry of vector Base. The vector contains the state of the binary tree search which specifies the row in TEMP currently associated with each (survived) edge.

**Step 2** computes *applies*. The step consists of $\lceil log(e) \rceil$ iterations each of which partitions the $e$ entries of the columns into areas of 2 sub-areas containing $2^{B-1}$ contiguous entries. By construction, only the areas which include entries referred to in Base can contain entries of edges in $E$. Hence only such areas are selected by step 2.2. Each time an area is selected, step 2.2.1 guarantees that the entry of the *survived* edge is put in the location of the lowest address in the area. This is accomplished by step 2.1 which sets Buddy to the address obtained by inverting, in Base[i], the B-th bit from the right. In this way Base[i] and its Buddy are the lowest addresses of the two sub-areas. Note that the lowest address in each selected area has the last B bits always equal to 0.

**Algorithm** SELECTS&APPLIES

**input:** E, Flag

**output:** E which is updated by selects and applies. Flag which is set to 0 on all *not-survived* indices and to 1 on the *survived* ones.

*Notes*: TEMP is an $e \times v$-entry array and supplies each source vertex of E with $e$ locations. Base is an $e$-entry array and contains the state of the binary tree search. Initially, TEMP[i,j]=0 for all $0 \leq i \leq e-1$ and all $1 \leq j \leq v$. $XOR\_bit(n,i)$ is the number resulting from inverting the i-th low bit of n.

```
begin
1.  Foreach i ∈ [1..e] Pardo
    begin
      if E[i,1] = E[i,2] then Flag[i]:=0 else TEMP[i-1,E[i,1]]:=i;
      Base[i]:= i-1
    end;
2.  For Inner:= 1 to ⌈log(e)⌉ do
      Foreach i ∈ [1..e] Pardo
        begin
2.1       Buddy:=XOR_bit (Base[i],Inner);
2.2       if Flag[i]≠ 0 and Buddy ≤ e then
            begin
              source:=E[i,1];
              index:=TEMP[Buddy,source];
2.2.1         if Buddy < Base[i] then
                if index ≠ 0 then Flag[i]:= 0
                else begin
                       swap(TEMP[Buddy,source],TEMP[Base[i,source]);
                       Base[i]:= Buddy
                  end
              else if index ≠ 0 then
                     begin
                       if E[i,2] > E[index,2]
                         then swap(E[i,2],E[index,2]);
                       E[index,1]:=E[index,2];
                       E[index,2]:=E[i,2];
                       TEMP[Buddy,source]:=0
                     end
            end
        end
end.
```

Figure 4: A program for SELECTS&APPLIES based on a binary tree search for edges selection and comparison

As far as the complexity of SELECTS&APPLIES is concerned, note that steps 1, 2.1, 2.2 require $O(1)$ time on the CREW model using $e$ processors. Hence we have the following.

**Theorem 2** *CONNECT computes the connected components of an undirected graph with $v$ vertices and $e$ edges in time $O(e * log^2(e)/p)$ on the CREW PRAM model using $p \leq e$ processors.*

# 8 Computing on EREW

Our algorithm can work with the same bound on the weakest memory shared EREW PRAM model [2]. Conflicts in reading can only occur during the *jumping* operated by step 5 of the main algorithm. This is due to the fact that distinct *survived* edges, which are collected by step 4, have different sources but may have the same target. We can find such edges by sorting all *survived* edges according to the target, i.e. $(x, y) \succ (x', y')$ iff either $y > y'$ or $y = y'$ *and* $x > x'$. Looking for adjacent edges in the sorted list, we can remove edges that share the target vertex by using an operation of *linearize* as described below. Let $(x_i, y_i)$ be the i-th edge of the list and $(x_{i+1}, y_{i+1})$ its right adjacent. *Linearize* replaces edge $(x_i, y_i)$ with the new edge $(x_i, x_{i+1})$ if and only if $y_i = y_{i+1}$. Hence, we preprocess step 4 in the main algorithm with step 4a below:

4a.1. *sort* the survived edges of $E$
  2. *linearize* the right adjacents

**Lemma 4** *Let $Surv$ be any list of* survived *edges and $Surv_L$ be the list resulting from the application of step 4a to $Surv$. Then, the jumping of $Surv$ computes the same values computed by the jumping of $Surv_L$.*

**Proof** Straightforward by induction.

*Sort* can be accomplished in $O((e/p)log(e))$ time on the EREW PRAM model using $p$ processors [6], while *linearize* in $O(e/p)$ time on the same model.

**Theorem 3** *CONNECT with the additional step 4a , put before step4, computes in time $O(e * log^2(e)/p)$ on the EREW PRAM model using $p \leq e$ processors.*

# 9 Conclusions and Related Topics

We have presented a new deterministic algorithm which computes CCug of a graph, with $e$ edges and $v$ vertices, in $0(log^2(e))$ time using $e$ processors. The algorithm cost, TS ($time \times processors$ product) is affected only by the number $e$ of the graph edges. Many algorithms have been proposed to compute CCug. An optimal linear solution for CCug on RAM serial machines is obtained using one of the two methods to search a graph, namely depth-search and breadth-search. At present, we know of no efficient deterministic algorithm to implement such methods on PRAM machines [15]. Again for serial machines, [28] furnishes an almost linear algorithm for CCug on graphs which grows dynamically. For CCug on PRAM, one of the first good solutions was presented in [14]. The algorithm was designed for the CREW model [21], computes in $O(log^2(v))$, and uses $v^2$ processors (also $v^2/log(v)$ with a different processor allocation strategy). The requirement of a number of processors which is quadratic in the number of graph vertices is also present in the algorithm proposed in [4]. This algorithm is, however, scalable on the number of processors which have to be allocated. A more efficient use of processors is in the algorithm in [26]. It is based on the approach of [14] and has the same depth but uses only $e + v\,log(v)$ processors. A fully scalable use of processor resources is provided in [12] where $p \leq e$ processors are used. Although, the time complexity of this algorithm increases up to $O(e/p + (v\,log(v))/p + log^2(v))$, it is relevant that the number of processors used in the algorithm, is proportional to the number of graph edges rather than vertices. Note that for dense graphs in which we have $e > O(v)$, the cost of the three above algorithms is the same. In particular, in the extreme case in which $p{=}e{=}O(v^2)$, the cost is $O(v^2 log^2(v))$. However, for sparse graphs, in which $e$ is increasing asymptotically faster than $v$, the algorithm in [12] is the best, since its TS is $O(v\,log(v) + e\,log^2(v))$. The cost improvement is of a factor (i.e. ratio TS'/TS, where TS is the cost of Han's algorithm and TS' is the cost of one of the other two) ranging in the interval $[v, v\,log(v)]$. CCug on sparse graphs has not a mere theoretical value. All these algorithms can be used to find the transitive closure of binary symmetric (and possibly, reflexive) relations. In this case, edges are the axioms of the relation while vertices form the range relation. Hence, using these algorithms the cost of computing a transitive closure is affected by the size of the range.

Based on the same *hooking technique*, another PRAM algorithm for CCug is in [27, 1]. In this algorithm, the time is reduced to $O(log(v))$ and the processors required are $2e+v$. However, the algorithm is for the strongest CRCW model which could suggest an additional time factor $log(v+e)$ in order to simulate it on the weak CREW model [29, 21]. According to this simulation, for dense graphs this algorithm has TS which is comparable to that of the three ones above, namely $O(e\,log^2(v))$. Moreover, for sparse graphs, the algorithm in [12] has again the best TS even though its improvement factor does not exceed $log(v)$. A slightly better performance was reached by the algorithm in [7], which shows an optimal processor bound when $e \geq v$ (i.e. for graphs which are not sparse). However, this algorithm too is for the CRCW model. Using a schedule technique to delay hookings, an algorithm for CCug on the EREW model was presented in [17]. The algorithm has depth $O(log^{1.5}(v))$ with $2e+v$ processors. The same bound is also obtained by [20, 25]. More recently, using a more sophisticated scheduling scheme, this bound has been significantly reduced to $O(log(v)log(log(v)))$ with $2e+v$ processors in [5]. This algorithm has the best known speedup and also the best TS for dense graphs. However, for sparse graphs the algorithm in [12] could still be competitive. In particular, even adding the cost of the simulation of CREW on the EREW model, the algorithm in [12] behaves better than the one in [5] by a factor $Log_e(log(v))$ when $v \gg e\,log(v)$, as is shown below:

$$
\begin{aligned}
&TS_{Han}[p=e] = O(e\,log(e) + v\,log(v)log(e) + e\,log(e)log^2(v)) \\
&TS_{Chong} = O((e+v)\,log(v)log(log(v))) \\
&TS_{Chong}/TS_{Han}[v \gg e\,log(v)] = O(v\,log(v)log(log(v)))/O(v\,log(v)log(e)) \\
&\qquad\qquad\qquad\qquad\qquad\quad = Log_e(log(v))
\end{aligned}
$$

where $TS_{Han}[p=e]$ is the cost of [Han 90] when $p=e$ processors are used, and $TS_{Chong}$ is the cost of [5]. In this case, our algorithm behaves better than both the above two as is shown below, where $TS_{Bell}[p=e]$ is the cost of our algorithm:

$$
\begin{aligned}
&TS_{Bell}[p=e] = O(e\,log^2(e) \\
&TS_{Han}/TS_{Bell}[v \gg e\,log(v)] = O(v\,log(v)log(e))/O(e\,log^2(e) \\
&\qquad\qquad\qquad\qquad\qquad\quad = (v/e)Log_e(v).
\end{aligned}
$$

One way to partially overcome the performance downfall that such algorithms have in dealing with sparse graphs is to preprocess graphs in order to localize and then, to remove the connected components that consist of isolated vertices. Using $v/log(v)$ processors, this can be accomplished in $O(log(v))$ time. The reduced graph is obviously a non sparse graph, in fact the requirement $e > v - 2$ holds. In the algorithm of [5] this use leads to a final cost, $TS_f$, which is the sum of the cost, $TS_{prep} = v$, due to the preprocessing of the input graph, and of the cost, $TS_{Chong} = O(e\, log(e) log(log(e)))$, due to applying the algorithm on the reduced graph. Compared with the previous costs, the technique yields an undeniable improvement. However, for $v \gg e\, log(e) log(log(e))$ the dominant cost is due to $TS_{prep}$ and it can be so high that the cost $TS_{Chong}$ can be disregarded. In this case too, our algorithm has the best cost.

The real innovation in our algorithm is in the approach, which resorts to a technique due to [13], and in the cost which is not affected by the number of vertices in the input graph. The last aspect is of no relevance when dealing with dense graphs and in this case it behaves like the algorithms in [14, 4, 27, 12]. We tested the performance of a quiet implementation of the algorithm on the *Connection Machine* and compared it with the ones in [14, 12]. The results in [2] show that our algorithm has a speed up factor ranging in the interval [1,2..1,57] even for dense graphs.

Our algorithm has been developed for the PRAM model. This model allows us to focus on the structure of the computational problem but hides many of the implementation issues of real machines. The compilation of PRAM programs into efficient machine code is still an open issue [23]. [8] shows that the algorithms for CCug, since then proposed (included the optimal algorithm in [27]), require a non trivial reformulation, in order to obtain reasonably efficient programs on the asynchronous APRAM model. This is due to the approach on which such algorithms are based. It would be interesting to carry out a similar investigation for the approach of the algorithm we propose. Algorithms for CCug on other kinds of machines [30, 19, 8, 3] have also been designed. All of them exhibit a computational structure which exploits various, specific, machine abilities. Among those, the algorithm in [19] is very competitive in time, since it is $O(1)$, but it is expensive in hardware size, since it requires the allocation of $v \times v$ processors whenever one connected component has to be accessed.

# References

[1] Awerbush B., and Shiloach, Y. New Connectivity and MSF algorithms for Shuffle-Exchange Network and PRAM. *IEEE Trans. on Comp.* c-36,

10 (1987), 1258-1263.

[2] Bellia M., Occhiuto, E., Performance of a EREW $O((e/p)log^2(e))$ depth $O(p)$ Processors Algorithm for Transitive Closure on the Connection Machine. In preparation (1997)

[3] Caceres E., Dehne F., Ferreira A., Flocchini P., Rieping I., Roncato A., Santoro N., Song S.W., Efficient Parallel Graphs Algorithms for Coarse Grained Mulicomputers. $24^{th}$ *Inter. Coll. ICALP'97, Automata Languages and Programs*, LNCS 1256, Degano, Gorrieri, Marchetti-Spaccamela Eds, 1997.

[4] Chin F. Y., Lam, J., and Chen, I. N. Efficient Parallel Algorithms for Some Graph Problems. *Comm. ACM* 25, 9 (1982), 659-665.

[5] Chong K. W., and Lam, T. W. Finding Connected Components in $O(\log n \log\log n)$ Time on the EREW PRAM. *J. Algorithms* 18, (1995), 378-402.

[6] Cole R. Parallel Merge Sort *SIAM J. Comp.* 17, 4 (1988) 770-785.

[7] Cole R. and Vishkin, U. Approximate Parallel Scheduling. Part II. Application to Optimal Parallel Graph Algorithms in Logarithmic Time, *Information and Computation* 92, 1 (1991) 1-47.

[8] Cole R. and Zajicek, O. An Asynchronous Algorithm for Undirected Graph Connectivity, *J. Algorithms* 18, (1995), 50-97.

[9] Cook S. A Taxonomy of Problems with fast parallel Algorithms. *Information and Control* 64 (1985) 2-22

[10] Downey P. J., Sethi, R., and Tarjan, R. E. Variations on the Common Sub expression Problem. *J. ACM* 27, 4 (1980), 758-771.

[11] Dwork C., Kanellakis, P. C., and Stockmeyer,L. Parallel Algorithms for Term Matching. *SIAM J. Comp.*17, 4 (1988), 711-731.

[12] Han Y., and Wagner, R. A. An Efficient and Fast Parallel-Connected Component Algorithm. *J. ACM* 37, 3 (1990), 626-642.

[13] Hirschberg D. S. Fast Parallel Sorting Algorithm. *Comm. ACM* 21, 10 (1978), 657-661.

[14] Hirschberg D. S., Chandra, A. K., and Sarwate D.V. Connected Components on Parallel Computers, *Comm. ACM* 22, 8 (1979), 461-464.

[15] Ja'Ja' J. An Introduction to Parallel Algorithms *Addison-Wesley*, (1992).

[16] Johnson D. S. A Catalog of Complexity Classes. In *Algorithms and Complexity*, van Leeuwen, J. Ed., Elsevier (1990), 67-162.

[17] Johnson D. S., and Metaxas P. Connected Components in $O(log^{3/2} \mid V \mid)$ parallel time for the CREW PRAM.*32nd Annual IEEE Symp. on Foundation of Computer Science*, (1991), 688-697

[18] Kanellakis P. C., and Revetz, P. Z.On the Relationship of Congruence Closure and Unification. In *Unification*, Kirchner, C. Ed., Academic Press, (1990), 139-156.

[19] Kao T.-W. , Horng, S.-J., and Tsai, H.-R. Computing Connected Components and Some Related Application on a RAP. *IEEE Int'l Conf. on Parallel Processing*, (1993), vol. III, 57-64.

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

[21] Karp R. M. and Ramachandram, V. Parallel Algorithms for Shared-Memory Machines. In *Algorithms and Complexity*, van Leeuwen, J. Ed., Elsevier (1990), 869-942.

[22] Kruskal C.P., Rudolph L. and Snir M.Efficient Parallel Algorithms for Graph problems. *Proc. of the Inter. Conf. on Parallel Processing*, (1986), 869-876.

[23] Kedem Z. M., Palem, K. V., Rabin, M.O. and Raghunathan, A. Efficient Program Transformation for Resilient Parallel Computation via Randomization. $24^{th}$ *Annual ACM Symposium on Theory of Computing*, (1992), 306-317.

[24] Nath N., Maheshwari, S. N. Parallel Algorithms for the Connected Components and Minimal Spanning Tree Problems. *Inf. Proc. Letters* 14, 1 (1982), 24-29.

[25] Nisan N., Szemeredi, E. and Wigderson, A. Undirected Connectivity in $O(log^{1.5} n)$ space. *33nd Annual IEEE Symp. on Foundation of Computer Science* (1992), 24-29.

[26] Savage C. and Ja'Ja', J. Fast, Efficient Parallel Alagorithms for some Graph Problems *SIAM J. Comp.*10, 4 (1981), 682-691.

[27] Shiloach Y. and Vishkin, U. An $O(log\, n)$ Parallel Connectivity Algorithm. *J. Algorithms* 3, 1 (1982), 57-67.

[28] Tarjan R. E. Efficiency of a good but not Linear Set Union Algorithm. *J. ACM* 22, 2 (1975), 215-225.

[29] Vishkin U. Implementation of Simultaneous Memory Address Access in Models that Forbid it. *J. Algorithms* 4, 1 (1983), 45-50.

[30] Wang B. F. and Chen, G.H. Constant Time Algorithms for the Transitive Closure and Some Related Graph Problems on Processor Arrays with a Reconfigurable Bus System. *IEEE Trans. on Parallel and Distributed Systems* 1, 4 (1990), 500-507.