Advanced Programming Techniques

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Connected components

Reminder:

- A connected component is an inclusion wise maximal set of vertices, any two of which can be linked by a path.
- A weak component of a directed graph is a connected component of its underlying graph (obtained by ignoring arc orientations).
- A strong component of a directed graph is an inclusion wise maximal set of vertices, any two of which can be linked by directed paths.

• Today's lecture: **parallel computation of connected components (and therefore, of weak components).**

Reminder: serial computation

```
**for every** vertex v:
 **if** v is unvisited:
  Start a DFS/BFS at v
  Assign label v to all visited vertices
  //we visit the connected component of v
```

Remark: the algorithm also computes a spanning tree for every component.

Issues to be resolved:

- graph searches BFS/DFS are difficult to parallelize!
- the main loop, over all vertices, cannot be easily parallelized (because two independent searches at different vertices may eventually join).

Minimum/Maximum Labelling

- A labelling of a graph $G = (V, E)$ is a function $f : V \to V$ such that:
  - $v, f(v)$ are in the same connected component;
  - u, v are in the same connected component if and only if $f(u) = f(v)$.

We sometimes called $f(v)$ the representative of the connected component of vertex v.

- Vertices in a graph may conveniently numbered by $1, 2, \ldots, n$. Then, a natural strategy consists in choosing as representative:
  - The vertex in the component of minimum number.
  - The vertex in the component of maximum number.

Parallel computation of min/max

Input: a doubly-linked list of numbers

Output: the minimum element in the list

Algorithm:

```
while there are $\geq 2$ elements:
 for every element e in parallel:
  if e has a predecessor e' and e' < e then erase e
  if e has a successor e' and e' $\leq$ e then erase e
```

Analysis: Let n denote the initial number of elements in the input.

- At every iteration, we halve the number of elements. Therefore, there are $\mathcal{O}(\log n)$ iterations.
- Step bound: $n/p + n/(2p) + n/(4p) + \ldots = n \log n/p$.

Application: resolving `write` conflicts

- Suppose that p concurrent processors try to overwrite some variable.

- This can be resolved using lock/mutex, but then the whole operation is serialized: it requires p computation steps.

- Instead, every processor has its own local version of the variable, and we use a reducer in order to select which local version we want to keep (this requires a synchronization mechanism such as a barrier).

- In case of number variables, we may select the minimum (or maximum) value. It can be computed in $\mathcal{O}(\log p)$ steps.

$\rightarrow$ This is known as "Combining CRCW".

The LT-framework

- Algorithms must maintain a rooted forest.
- Edges of the forest have no real existence (*i.e.*, they may be not edges of the graph).
- However, **all nodes in the same rooted tree must belong to the same connected component**.

Encoding: Every vertex v has a field $v.p$ such that, at any moment during the algorithm:

- $v.p = v$ if and only if v is a root;
- if $v \neq v.p$, then $v.p$ is the father of v in the rooted forest (but not necessarily in the graph itself).

Remark: acylicity must be guaranteed by the algorithm (it would be too costly to check it).

The LT-framework

Initialization

- Initially, $v.p = v$ for every vertex v. (Every vertex is a root).

- At the very end of the procedure, the rooted forest must satisfy the following properties:
  - every connected component is fully contained in one rooted tree (there are as many trees as connected components).
  - every rooted tree is a star (so that every vertex has access in constant-time to the representative of its component).

Important remark: we do NOT get spanning trees for the connected components.

The LT-framework

Main loop

- We repeat the following phases while at least one vertex *v* modifies its parent node *v*.*p*:

  - `connect`: each *v* receives parent proposals from its current neighbours.
  - `shorcut`: the heights of non-star rooted trees are shrunk (roughly, halved).
  - `alter`: edges of the graph may be modified. This must be done without changing the sets of connected components.

→ There are several implementations for each phase, which lead to several different connected component computation algorithms.

Connect step: `direct-connect`

- We use the minimum of v and w as a candidate for the new parent of the other.

- The parent is changed only if the label decreases (recall that we want a minimum labelling).

```
direct-connect:
    for each edge $\{v, w\}$ do
        if $v > w$ then
            $v.p = \min\{v.p, w\}$
        else $w.p = \min\{w.p, v\}$
```

Requires handling conflicts with Combine CRCW.

Connect step: `parent-connect`

- We attempt to modify grandparents (either $v.p.p$ or $w.p.p$), and not directly parents.
- As before, parent is changed only if the label decreases.

```
*parent-connect*:
    **for** each vertex $v$ **do**
        $v.o = v.p$
    **for** each edge $\{v, w\}$ **do**
        **if** $v.o > w.o$ **then**
            $v.o.p = \min\{v.o.p, w.o\}$
        **else** $w.o.p = \min\{w.o.p, v.o\}$
```

Requires handling conflicts with Combine CRCW

Remark: $v.p$ may

Variations: preserving the trees

- Previous approaches may move one subtree (e.g., rooted at either v or $v.p$) to another rooted tree of the forest (e.g., the one containing w).

- Intuitively, it is desirable to *merge* rooted trees, to quickly find connected components. This can be achieved if we only allow to modify parent points for *roots*.

```
direct-root-connect:
    for each vertex $v$ do
        $v.o = v.p$
    for each edge $\{v, w\}$ do
        if $v > w$ and $v = v.o$ then
            $v.p = \min\{v.p, w\}$
        else if $w = w.o$ then
            $w.p = \min\{w.p, v\}$
```

```
parent-root-connect:
    for each vertex $v$ do
        $v.o = v.p$
        for each edge $\{v, w\}$ do
            if $v.o > w.o$ and $v.o = v.o.o$ then
                $v.o.p = \min\{v.o.p, w.o\}$
            else if $w.o = w.o.o$ then
                $w.o.p = \min\{w.o.p, v.o\}$
```

Requires handling conflicts with Combine CRCW

Shortcut step

- Every vertex selects its grandparent as its new parent node.

Remark: roots and their respective children are left unchanged.

```
shortcut:
    for each vertex $v$ do
        $v.o = v.p$
    for each vertex $v$ do
        $v.p = v.o.o$
```

$\rightarrow$ Barrier needed between the two for loops.

Alter step

- Edges "climb up" in the rooted tree.

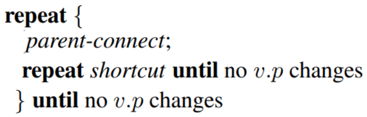
Remark: preserves the connected components of the graph.

- This is especially useful for connect step implementation where we only allow to modify the parents of roots.

```
alter:
    for each edge $\{v, w\}$ do
        if $v.p = w.p$ then
            delete $\{v, w\}$
        else replace $\{v, w\}$ by $\{v.p, w.p\}$
```

- We implicitly assume that we can only merge rooted trees, i.e., we never move one subtree to another rooted tree (otherwise, altering the edges could prevent us from finding the connected components).

Algorithm S



```
repeat {
  parent-connect;
  repeat shortcut until no $v.p$ changes
} until no $v.p$ changes
```

- At the beginning of every iteration, every rooted tree must be a star (this is because of repeated iterations of shortcut steps).

- Then, with `parent-connect` we can merge rooted stars together $\rightarrow$ we only merge rooted trees together.

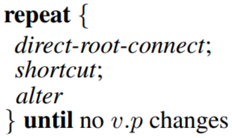
Analysis:

Each iteration requires $\mathcal{O}(\log n)$ calls to `shortcut`.

Every rooted tree is merged after at most two iterations. Therefore, the number of iterations is in $\mathcal{O}(\log n)$.

Step bound: $(m+n)\log^2 n/p$.

Algorithm RA



```
repeat {
    direct-root-connect;
    shortcut;
    alter
} until no $v.p$ changes
```

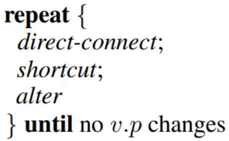
- We can only merge trees together. In particular, the number of trees is nonincreasing.

- All edges climb up to the root in $\mathcal{O}(\log n)$ iterations.

$\rightarrow$ we can prove this way a step bound in $(m + n) \log^2 n/p$.

However, this is not tight. The right bound is $(m + n) \log n/p$.

Algorithm A



```
repeat {
  direct-connect;
  shortcut;
  alter
} until no $v.p$ changes
```

- Let r be the minimum vertex of a connected component.

- After one iteration, r becomes the parent of all its neighbours (because of `direct-connect`).

- Furthermore, r becomes adjacent to all its vertices at distance two (because of `alter`).

Consequence: Termination in D rounds, where D is the largest diameter of a connected component.

Construction of a spanning tree

- Recall that rooted trees manipulated by the algorithm have no real existence. **They are not (partial) spanning trees!**

- However, for Algorithms S and RA, we can construct for each rooted tree T a (real) spanning tree in G for $V(T)$. Indeed, whenever we merge two rooted trees T, T', there exists a real edge between $V(T)$ and $V(T')$, which we add to the spanning tree of their component.

- Note that for Algorithm RA, we need to keep track of original edges (because of `alter`).

2-connected components

- A graph is 2-connected if its is connected, and it stays connected if we remove any of its vertices.

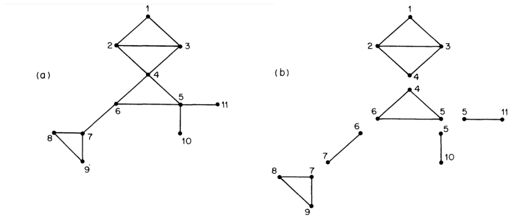
- A cut-vertex is one whose removal disconnects the graph.

- A 2-connected component of a graph, also known as a **block**, is a 2-connected subgraph that is maximal by inclusion.

- There is a serial $\mathcal{O}(n+m)$ time algorithm for computing all 2-connected components of a graph. Unfortunately, it relies on DFS, for which no efficient parallelization is in sight…

Beyond DFS: edge equivalence classes

- Blocks do NOT partition the vertices: this is because cut-vertices are in at least two blocks.
- **However, blocks do partition edges**. We call two edges e_1 and e_2 equivalent, denoted by $e_1 R e_2$, if they are edges in the same block.
- An edge is a *bridge* if it is not equivalent to any other edge.



Objective: compute edge equivalence classes.

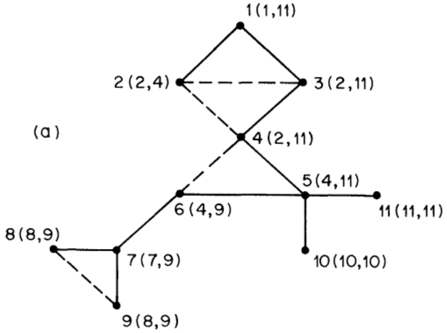
From blocks to connected components

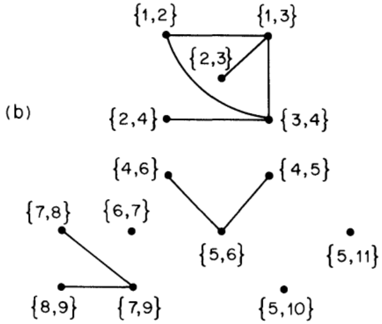
- Compute *any* spanning tree T of G (not necessarily a DFS tree).
- **Compute a DFS of T (not of G)**. Nodes are preordered from 1 to n.
- Construct a new graph G' from T, whose vertices are exactly G edges.

Two edges e, e' are adjacent in G' if:

- $e = \{v, v.p\}$ is a T edge, and $e' = \{v, w\}$ is a *backward edge* (w is an ancestor of v).
- $e = \{v, v.p\}$ and $e' = \{w, w.p\}$ are T edges, $\{v, w\}$ is an edge of G, and v, w are unrelated in T (cannot happen if T were a DFS tree).
- $e = \{v, v.p\}$ and $\{w, v = w.p\}$ are two consecutive T edges, and there is an edge with one end in T_w (subtree rooted at w) and the other in $T \setminus T_v$.

Example


(a)
1(1,11)
2(2,4)
3(2,11)
4(2,11)
5(4,11)
6(4,9)
11(11,11)
8(8,9)
7(7,9)
10(10,10)
9(8,9)


(b)
{1,2}
{1,3}
{2,3}
{2,4}
{3,4}
{4,6}
{4,5}
{7,8}
{6,7}
{5,6}
{5,11}
{8,9}
{7,9}
{5,10}

Correctness

Theorem (Tarjan & Vishkin, 85)

The connected components of G' are exactly the edge equivalence classes of G. In particular, we obtain the blocks of G from the connected components of G'.

Short justification:

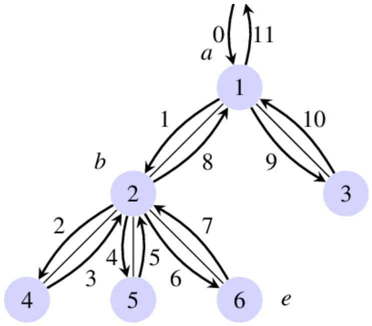
- Every edge of $G \setminus T$ closes a cycle, with all other edges in T. This is called a fundamental cycle.
- All edges of a fundamental cycle are in the same connected components of G'.
- Every other cycle can be seen as a "sum" of fundamental cycles (we repeatedly glue two such cycles, and discard common edges).

Parallel DFS on a tree

Euler Tour technique

Preliminary Remark: every edge $\{v, v.p\}$ must be visited twice during a DFS (once for visiting v, then one more time to backtrack to $v.p$).

- If we replace every edge $\{v, v.p\}$ by two **arcs** $(v.p, v), (v, v.p)$, then a DFS execution now corresponds to an Euler tour (every arc is visited exactly once).



Parallel construction of an Euler tour

- It suffices to decide, for every arc (either $(v.p, v)$ or $(v, v.p)$), which successor it has on the Euler tour.
- Case of an arc $(v.p, v)$: if v is a leaf, then the successor must be $(v.p, v)$. Otherwise, the successor is (v, u_1), where u_1 is the first child of v (for some fixed arbitrary order of children, e.g. by index).
- Case of an arc $(v, v.p)$: three subcases.
  - if w is the next child of $v.p$, then the successor must be $(v.p = w.p, w)$.
  - If v is the last child of $v.p$, and $v.p$ is not the root, then the successor must be $(v.p, v.p.p)$.
  - If $v.p$ is the root, u its first child, and v its last child, then the successor must be $(v.p, u)$.

Step bound: $\mathcal{O}(n/p)$**.**

Doubling technique

Input: a list with n elements

Output: for all elements e in the list, and $0 \leq i \leq \lfloor \log n \rfloor$, compute $p_i(e)$: a pointer to its 2^ith predecessor in the list, if it exists (or NULL).

$\rightarrow$ We may see the p_i's as "shortcuts" to quickly navigate in a list.

Algorithm:

```
for i = 0, ...log(n)
 for all elements e in parallel
  if i == 0
   $p_i(e)$ := pointer to $e$ predecessor in the list
  else
    $p_i(e)$ := $p_{i-1}(p_{i-1}(e))$
```

Step bound: $n \log n / p$.

Positions in a list

Proposition: given a list with n elements, we can compute in parallel the indices/positions of every element, in $\mathcal{O}(n \log n/p)$ steps.

```
for all elements $e$ in parallel
 pos(e) := 0
 i := 0, j:= 1
 while $p_i(e) \neq$ NULL { i := i+1, j := j*2 }
 r := e
 while $i \geq 0$
  if $p_i(r) \neq$ NULL { pos(e) := pos(e) + j, r := $p_i(r)$ }
  i := i-1, j := j/2
```

Intuition: we compute the binary representation of each index.

Applications

Consider the Euler Tour computed for T (spanning tree) as a list:

- For every node v different from the root, we can compute the size of the subtree rooted at v, simply by comparing the respective positions of the arcs $(v.p, v)$ and $(v, v.p)$.

- Remove from the Euler tour (in parallel) all backward arcs $(x, x.p)$. Then, for every node v different from the root, we can compute its preorder, simply by computing the position of $(v.p, v)$ in the list.

Reminder (?): in a preordering, all nodes in a subtree must be consecutive. In particular, every node v knows its rooted subtree, under the form of a preordering interval $[b_v, f_v]$ (with b_v the preorder of v).

Parallel computation of G' (1/2)

Recall that edges $\{e, e'\}$ of G' are of three different types:

- Type 1: $e = \{v, v.p\}$ is a T edge, and $e' = \{v, w\}$ is a *backward edge*.

$\Longrightarrow$ Since we have access to $[b_w, f_w]$ (subtree rooted at w), we can detect this case when we scan all edges $\{v, w\}$ of G.

- Type 2: $e = \{v, v.p\}$ and $e' = \{w, w.p\}$ are T edges, $\{v, w\}$ is an edge of G, and v, w are unrelated in T.

$\Longrightarrow$ Since we have access to $[b_v, f_v]$ and $[b_w, f_w]$, we can also detect this case when we scan all edges $\{v, w\}$ of G.

- Type 3: $e = \{v, v.p\}$ and $\{w, v = w.p\}$ are two consecutive T edges, and there is an edge with one end in T_w (subtree rooted at w) and the other in $T \setminus T_v$.

$\Longrightarrow$ Harder to detect, because non local

Low and High

<u>Reminder</u>: every vertex has access to its rooted subtree in T as a preorder interval $[b_v, f_v]$ (with b_v the preorder of v).

- We define:
  - `locallow[v]`: the minimum preorder value b_u among all neighbours u of v <u>in G</u>.
  - `low[v]`: the minimum of `locallow[x]` among all descendants x of v <u>in T</u>.
  - `localhigh[v]` and `high[v]` are defined similarly, by taking maxima rather than minima.

Key observation: there is an edge with one end in T_w (subtree rooted at w) and the other in $T \setminus T_v$ if and only if: `low[w]` $< b_v$, or `high[w]` $> f_v$.

Range minimum queries (in parallel)

Problem considered: preprocess a vector `v[]` such that we can answer as fast as possible to the following type of queries:

$$\texttt{q(i,j):}\quad \texttt{compute}\ \min\{\ \texttt{v[k]}\ :\ \texttt{i} \leq \texttt{k} \leq \texttt{j}\ \}$$

Solution:

- Precompute the answer for all queries $q(i, j = i + 2^b)$, where $b \in \{0, 1, \ldots, \lfloor \log n \rfloor$. It can be done in $\mathcal{O}(n \log n / p)$ steps by using doubling technique.
- Every interval $[i, j]$ can be decomposed in $\mathcal{O}(\log n)$ intervals $[i', i' + 2^b]$ for which we already precomputed the solution.

Parallel computation of G' (2/2)

- The values `locallow[v]` and `localhigh[v]` can be computed in parallel in $\mathcal{O}(m/p)$ steps, if we traverse the adjacency lists.

- The values `low[v]` and `high[v]` can be reduced to range minimum queries (in parallel) on the Euler tour. In particular, they can be computed in $\mathcal{O}(n \log n/p)$ steps.

Consequence: we can compute G' in parallel in $\mathcal{O}((m + n \log n)/p)$ steps.

Theorem

We can compute all the blocks in parallel, in $(n + m) \log n/p$ steps.

Questions