Advanced Programming Techniques

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

Weak and Strong Components

Reminder: Connected components (and so, weak components of directed graphs) can be efficiently computed in parallel.

$\rightarrow$ The same is true for blocks (2-connected components).

- However, the situation looks much more complicated for **strong components**: the best known serial algorithms (Tarjan, Kosaraju, etc.) are built on DFS, which (we think) cannot be parallelized.

- Today's lecture: **parallel computation of strong components.**

$\rightarrow$ Reduction to *Union-Find* operations.

Disjoint Sets

Reminder

A Disjoint Sets Data Structure maintains a collection of pairwise disjoint sets. It supports the following three basic operations:

- `makeset(x)`: If x is not already present in the collection, then add a new singleton set whose unique elements equals x.
- `find(x)`: outputs the unique identifier of the set containing x.

  $\rightarrow$ In general, `find(x)` outputs an element of the set, also called its "representative".
  $\rightarrow$ In serial implementations, we may force this representative to have special properties (*e.g.*, largest element in the set) with no computational overhead.
- `union(x,y)`: merge the respective sets of x,y into one.

Relation with connected components

Consider the following algorithm on a graph G:

for all `vertices v` **in parallel**
 `makeset(v)`

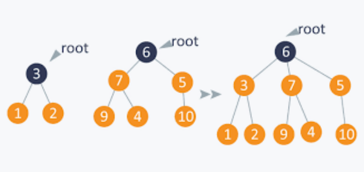
for all `edges uv` **in parallel**
 `union(u,v)`

for all `vertices v` **in parallel**
 `cc[v] = find(v)`

Consequence: a parallel implementation for Disjoint Sets leads to a new parallel implementation for connected components computation.

Serial implementation: Representing sets as **trees**

- The elements of each set are the nodes of a tree, whose root is the representative of this set.



- `makeset(x)`: create a new tree whose unique node is x
- `find(x)`: find the root of the tree containing x as a node (climb up)
- `union(x,y)`: link together the trees that contain x, y as nodes (one root becomes the child of the other).

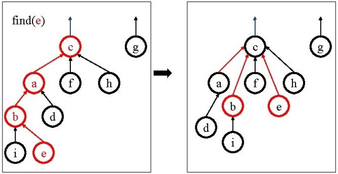
In a naive implementation, complexity depends on the height of the trees.

Improvements: Path Compression

Operation `find`

In order to access to the root (representative), we climb in the tree. *On our way, all visited nodes are reconnected as children of the root*.

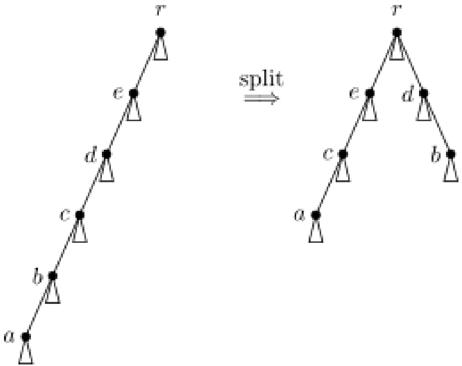
→ Speed-up of subsequent `find` operations.



Improvements: Path Splitting

Operation `find`

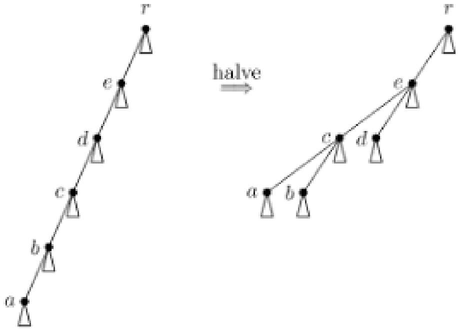
All nodes on the path to the root are reconnected to their grandparent.



Improvements: Path Halving

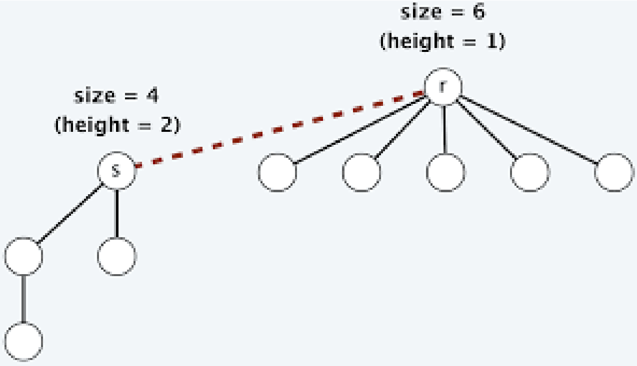
Operation `find`

On our way to the root, we reconnect each traversed node to its grandparent. Former parent nodes are skipped.



Improvements: Union by size

• Each node stores the size of its rooted subtree. If we merge two sets, then the root of the new set is the root of the biggest tree.

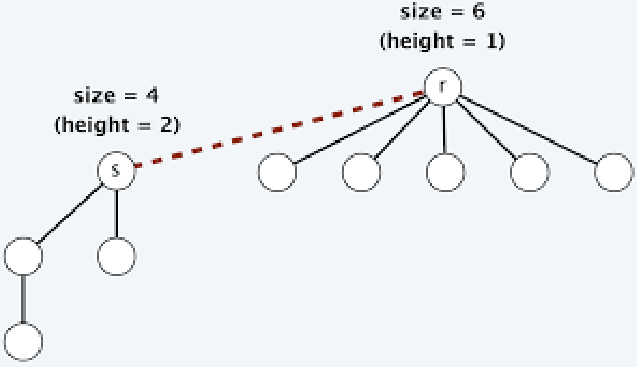


Remark: Ensures logarithmic depth.

Improvements: Union by rank

- Each node keeps a rank: **upper bound** on its depth. If we merge two sets, then the root of the new set is the root of larger rank.

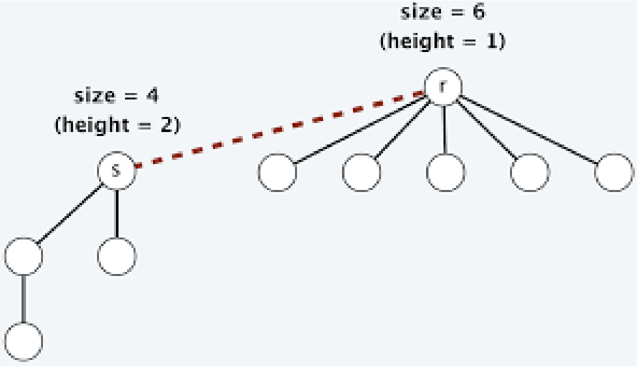
→ The real depth might be smaller than the rank because of path compression/splitting/halving...



Also ensures logarithmic depth.

Improvements: Union by **random index**

- Node first select random <u>distinct</u> indices. If we merge two sets, then the root of the new set is the root of larger (or smaller) index.



Also ensures logarithmic depth (in expectation).

Computation of a random index

Assumption: the universe of n elements is fixed, and known to us from the beginning.

- Every element just selects u.a.r. a number between 1 and n^3.

- For any fixed element, the probability for another element to select its same index is at most $n \times 1/n^3 = 1/n^2$.

- Therefore, all indices are distinct with probability at least $1 - 1/n$.

Remark: can be done in parallel (assuming the random seed has no correlation with our scheduler).

Toward a parallel implementation: The `CAS` primitive

`CAS(x, y, z)`:

Input: an adress x in the shared memory; two values y,z.

Output: `true` if x was containing value y before the operation (after the operation, x stores the new value z); `false` otherwise.

This is an **atomic** compare-and-swap operation (it requires synchronization).

- OpenMP implementation:

```
#pragma omp atomic compare
if(x == y) x = z;
```

Concurrent Disjoint Sets

The need for more operations

- In a serial implementation, we only have three operations: `makeset`, `find`, `union`.

- A `union` can be seen as two `find` operations (that can be done in parallel), followed by a `link` operation between two roots. In a concurrent setting, a `link` can sometimes fail! Therefore, we need to define `link` separately from `union`, and to repeatedly call `link` until it becomes successful.

- We mostly rely on `find` in order to decide whether two elements are in the same set. However, this approach may fail in a concurrent setting. Therefore, we need to define a `same_set` operation separately.

Link

- Linking by size or rank is problematic because of concurrent operations. However, a simple implementation can still be achieved using Linking by random index.

1: **procedure** *link*(u, v)
2: **if** $u < v$ **then** $\mathrm{CAS}(u.p, u, v)$
3: **else** $\mathrm{CAS}(v.p, v, u)$

<u>Remark</u>: the operation fails if $u \neq u.p$, *i.e.*, u is no more a root because of concurrent operations.

Find (1/2)

- Path compaction (reconnection of nodes on the path to the root) may fail in a concurrent setting.

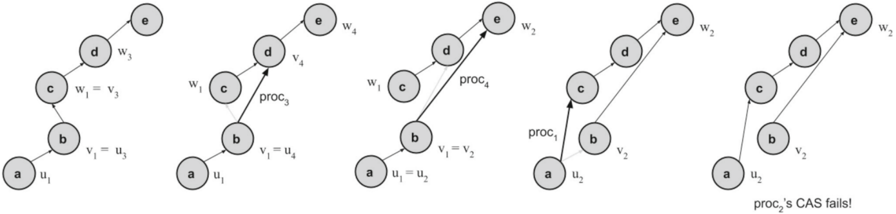
- However, "*intuitively*", when it does fail, it means that compaction has been done by another processor.

$\Longrightarrow$ we continue climbing up into the tree *even if the compaction has failed*!

procedure *find*(x)
    $u \leftarrow x$; $v \leftarrow u.p$; $w \leftarrow v.p$
    while $v \neq w$ **do**
        CAS$(u.p, v, w)$; $u \leftarrow v$; $v \leftarrow u.p$; $w \leftarrow v.p$
    return v

Find (2/2)

Example



Same_set

- Concurrent calls to `find` may output two different nodes, even if they belong to the same set.

- We repeatedly need to check whether the root has changed since our last call to `find`.

procedure *same-set*(x, y)
    $u \leftarrow \mathit{find}(x);\ v \leftarrow \mathit{find}(y)^*$
    while $u \neq v$ **do**
        $w \leftarrow u.p$
        if $u = w$ **then return** false
        $u \leftarrow \mathit{find}(u);\ v \leftarrow \mathit{find}(v)^*$
    return true

Union

- Repeated calls to `link` until it becomes successful.

```
procedure unite(x, y)
    u ← find(x); v ← find(y)
    while u ≠ v do
        link(u, v)
        u ← find(u); v ← find(v)
```

Strong components

Purdom-Monroe Algorithm

```
∀v ∈ V : 𝒮(v) := {v}
DEAD := VISITED := ∅
R := ∅
SETBASED(v_0)
procedure SETBASED(v)
    VISITED := VISITED ∪ {v}
    R.PUSH(v)
    for each w ∈ POST(v) do
        if w ∈ DEAD then continue ....... [already completed SCC]
        else if w ∉ VISITED then ................ [unvisited node w]
            SETBASED(w)
        else while 𝒮(v) ≠ 𝒮(w) do .................. [cycle found]
            r := R.POP()
            UNITE(𝒮, r, R.TOP())
    if v = R.TOP() then ............... [completely explored SCC]
        report SCC S(v)
        DEAD := DEAD ∪ S(v)
        R.POP()
```

- *Partial* strong components are stored. This is $\neq$ from Tarjan's algorithm where each strong component is fully computed at once.

Strong components

Purdom-Monroe Algorithm

$\forall v \in V : \mathcal{S}(v) := \{v\}$ makeset(v)
DEAD := VISITED := $\emptyset$
$R := \emptyset$
SETBASED(v_0)
procedure SETBASED(v)
    VISITED := VISITED $\cup \{v\}$
    R.PUSH(v)
    for each $w \in$ POST(v) **do**
        if $w \in$ DEAD **then continue**
        else if $w \notin$ VISITED **then**
            SETBASED(w)
        else while $\mathcal{S}(v) \neq \mathcal{S}(w)$ **do** same_set(u,v)
            $r := R$.POP()
            UNITE($\mathcal{S}, r, R$.TOP()) union
    if $v = R$.TOP() **then**
        report SCC $S(v)$
        DEAD := DEAD $\cup S(v)$
        R.POP()

- *Partial* strong components are stored. This is $\neq$ from Tarjan's algorithm where each strong component is fully computed at once.

UFSCC Algorithm

A modified Purdom-Monroe Algorithm is executed in parallel by all processors, with one shared Disjoint Sets data structures for partial strong components.

```
∀v ∈ V : S(v) := {v} ................................ [global S]
DEAD := DONE := ∅ ..................... [global DEAD and DONE]
∀p ∈ [1 ... p] : R_p := ∅ ............................ [local R_p]
UFSCC_1(v_0) ‖ ... ‖ UFSCC_p(v_0)
procedure UFSCC_p(v)
    R_p.PUSH(v)
    while v′ ∈ S(v) \ DONE do
        for each w ∈ RANDOM(POST(v′)) do
            if w ∈ DEAD then continue ..................... [DEAD]
            else if ∄w′ ∈ R_p : w ∈ S(w′) then ............. [NEW]
                UFSCC_p(w)
            else while S(v) ≠ S(w) do ..................... [LIVE]
                r := R_p.POP()
                UNITE(S, r, R_p.TOP())
        DONE := DONE ∪ {v′}
    if S(v) ⊈ DEAD then DEAD := DEAD ∪ S(v); report SCC S(v)
    if v = R_p.TOP() then R_p.POP()
```

UFSCC Algorithm

A modified Purdom-Monroe Algorithm is executed in parallel by all processors, with one shared Disjoint Sets data structures for partial strong components.

$\forall v \in V : S(v) := \{v\}$
DEAD := DONE := $\emptyset$
$\forall p \in [1 \ldots p] : R_p := \emptyset$
$\text{UFSCC}_1(v_0) \| \ldots \| \text{UFSCC}_p(v_0)$
procedure $\text{UFSCC}_p(v)$
  R_p.PUSH(v)
  while $v' \in S(v) \setminus$ DONE **do**
    for each $w \in$ RANDOM(POST(v')) **do**
      if $w \in$ DEAD **then continue**
      else if $\nexists w' \in R_p : w \in S(w')$ **then**
        $\text{UFSCC}_p(w)$
      else while $S(v) \neq S(w)$ **do**
        $r := R_p$.POP()
        UNITE(S, r, R_p.TOP())
    DONE := DONE $\cup \{v'\}$
  if $S(v) \not\subseteq$ DEAD **then** DEAD := DEAD $\cup\, S(v)$; **report SCC** $S(v)$
  if $v = R_p$.TOP() **then** R_p.POP()

repeated iterations over the partial strong component of v

(because of parallelism)

UFSCC Algorithm

A modified Purdom-Monroe Algorithm is executed in parallel by all processors, with one shared Disjoint Sets data structures for partial strong components.

```
∀v ∈ V : S(v) := {v}
DEAD := DONE := ∅
∀p ∈ [1 ... p] : R_p := ∅
UFSCC_1(v_0) || ... || UFSCC_p(v_0)
procedure UFSCC_p(v)
    R_p.PUSH(v)
    while v' ∈ S(v) \ DONE do
        for each w ∈ RANDOM(POST(v')) do
            if w ∈ DEAD then continue
            else if ∄w' ∈ R_p : w ∈ S(w') then
                UFSCC_p(w)
            else while S(v) ≠ S(w) do
                r := R_p.POP()
                UNITE(S, r, R_p.TOP())
        DONE := DONE ∪ {v'}
    if S(v) ⊈ DEAD then DEAD := DEAD ∪ S(v); report SCC S(v)
    if v = R_p.TOP() then R_p.POP()
```

w' was pushed in the stack before v. Therefore, v and w are in the same scc.

UFSCC Algorithm

A modified Purdom-Monroe Algorithm is executed in parallel by all processors, with one shared Disjoint Sets data structures for partial strong components.

```
∀v ∈ V : S(v) := {v}
DEAD := DONE := ∅
∀p ∈ [1 . . . p] : R_p := ∅
UFSCC_1(v_0)‖ . . . ‖UFSCC_p(v_0)
procedure UFSCC_p(v)
    R_p.PUSH(v)
    while v′ ∈ S(v) \ DONE do
        for each w ∈ RANDOM(POST(v′)) do
            if w ∈ DEAD then continue
            else if ∄w′ ∈ R_p : w ∈ S(w′) then
                UFSCC_p(w)
            else while S(v) ≠ S(w) do
                r := R_p.POP()
                UNITE(S, r, R_p.TOP())          critical section
        DONE := DONE ∪ {v′}
    if S(v) ⊈ DEAD then DEAD := DEAD ∪ S(v); report SCC S(v)
    if v = R_p.TOP() then R_p.POP()
```

Some Terminology

For a vertex v in a directed graph G, let us define:

- `pred(G,v)`: all vertices that can reach v in G
- `desc(G,v)`: all vertices that can be reached from v in G

Some Terminology

For a vertex v in a directed graph G, let us define:

- `pred(G,v)`: all vertices that can reach v in G
- `desc(G,v)`: all vertices that can be reached from v in G

Proposition 1: the strong component of v is exactly `pred(G,v)` $\cap$ `desc(G,v)`.

Some Terminology

For a vertex v in a directed graph G, let us define:

- `pred(G,v)`: all vertices that can reach v in G
- `desc(G,v)`: all vertices that can be reached from v in G

Proposition 1: the strong component of v is exactly `pred(G,v)` $\cap$ `desc(G,v)`.

Proposition 2: every strong component must either be fully in `pred(G,v)`, be fully in `desc(G,v)`, or not intersect `pred(G,v)` $\cup$ `desc(G,v)`.

Some Terminology

For a vertex v in a directed graph G, let us define:

- `pred(G,v)`: all vertices that can reach v in G
- `desc(G,v)`: all vertices that can be reached from v in G

Proposition 1: the strong component of v is exactly `pred(G,v)` $\cap$ `desc(G,v)`.

Proposition 2: every strong component must either be fully in `pred(G,v)`, be fully in `desc(G,v)`, or not intersect `pred(G,v)` $\cup$ `desc(G,v)`.

Remark: `pred(G,v)` and `desc(G,v)` can be computed using BFS (no need for DFS).

Forward-Backward Algorithm

procedure FWBW(G)
  if $G = \varnothing$ **then**
    return $\varnothing$
  select pivot v
  $D \leftarrow DESC(G, v)$
  $P \leftarrow PRED(G, v)$
  $R \leftarrow (G \setminus (P \cup D)$
  $S \leftarrow (D \cap P)$
  $FWBW(D \setminus S)$
  $FWBW(P \setminus S)$
  $FWBW(R)$

Forward-Backward Algorithm

procedure FWBW(G)
  if $G = \varnothing$ **then**
    return $\varnothing$
  select pivot v
  $D \leftarrow DESC(G, v)$
  $P \leftarrow PRED(G, v)$
  $R \leftarrow (G \setminus (P \cup D))$
  $S \leftarrow (D \cap P)$
  $FWBW(D \setminus S)$
  $FWBW(P \setminus S)$
  $FWBW(R)$

Two graph searches (BFS) **in parallel**

Forward-Backward Algorithm

procedure FWBW(G)
    if $G = \varnothing$ **then**
        return $\varnothing$
    select pivot v
    $D \leftarrow DESC(G, v)$
    $P \leftarrow PRED(G, v)$
    $R \leftarrow (G \setminus (P \cup D))$
    $S \leftarrow (D \cap P)$
    $FWBW(D \setminus S)$
    $FWBW(P \setminus S)$
    $FWBW(R)$

Can be computed in parallel

Forward-Backward Algorithm

procedure FWBW(G)
    if $G = \varnothing$ **then**
        return $\varnothing$
    select pivot v
    $D \leftarrow DESC(G, v)$
    $P \leftarrow PRED(G, v)$
    $R \leftarrow (G \setminus (P \cup D)$
    $S \leftarrow (D \cap P)$
    $FWBW(D \setminus S)$
    $FWBW(P \setminus S)$
    $FWBW(R)$

In parallel

Coloring Algorithm

- The graph is vertex coloured. Each colour c is 'owned" by a vertex (the one c_v of which c represents the identifier).

- Key invariant: all vertices with same colour c are contained in `desc(G,`c_v`)`. Furthermore, no vertex of `desc(G,`c_v`)` with another colour than c can be in the same strong component as c_v.

while $G \neq \varnothing$ **do**
    initialize $colors(v_{id}) = v_{id}$
    while at least one vertex has changed colors **do**
        for all $v \in G$ **do**
            for all $u \in N(v)$ **do**
                if $colors(v) > colors(u)$ **then**
                    $colors(u) \leftarrow colors(v)$
    for all unique $c \in colors$ **do**
        $SCC(c_v) \leftarrow PRED(G(c_v), c)$
        $G \leftarrow (G \setminus SCC(c_v))$

Coloring Algorithm

- The graph is vertex coloured. Each colour c is 'owned" by a vertex (the one c_v of which c represents the identifier).

- Key invariant: all vertices with same colour c are contained in $\texttt{desc}(G, c_v)$. Furthermore, no vertex of $\texttt{desc}(G, c_v)$ with another colour than c can be in the same strong component as c_v.

while $G \neq \varnothing$ **do**
    initialize $colors(v_{id}) = v_{id}$
    while at least one vertex has changed colors **do**
        for all $v \in G$ **do**
            for all $u \in N(v)$ **do**
                if $colors(v) > colors(u)$ **then**
                    $colors(u) \leftarrow colors(v)$
    for all unique $c \in colors$ **do**   In parallel
        $SCC(c_v) \leftarrow PRED(G(c_v), c)$
        $G \leftarrow (G \setminus SCC(c_v))$

Coloring Algorithm

- The graph is vertex coloured. Each colour c is 'owned" by a vertex (the one c_v of which c represents the identifier).
- Key invariant: all vertices with same colour c are contained in `desc(G,`c_v`)`. Furthermore, no vertex of `desc(G,`c_v`)` with another colour than c can be in the same strong component as c_v.

while $G \neq \emptyset$ **do**
  initialize $colors(v_{id}) = v_{id}$
  while at least one vertex has changed colors **do**
    for all $v \in G$ **do**
      for all $u \in N(v)$ **do**
        if $colors(v) > colors(u)$ **then**
          $colors(u) \leftarrow colors(v)$
  for all unique $c \in colors$ **do**
    $SCC(c_v) \leftarrow PRED(G(c_v), c)$
    $G \leftarrow (G \setminus SCC(c_v))$

In parallel

Questions