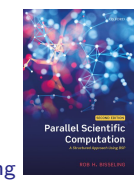
Suitors and Sorting

Section 5.3 of Parallel Scientific Computation, 2nd edition

Rob H. Bisseling

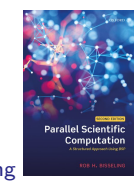
Utrecht University



Data structure for the graph

- A data structure stores all the information about the vertices and edges of the graph needed to run the matching algorithm.
- Let $v_0, \ldots, v_{n-1}$ be the vertices and $e_0, \ldots, e_{m-1}$ be the edges.
- For an edge $e = (v_i, v_j)$, we store the weight $\omega(e)$.
- For a vertex $v = v_i$, we store:
  - $pref(v)$, the preferred partner, i.e., the neighbouring vertex still available with the highest connecting edge weight;
  - $suitor(v)$, the suitor, i.e., the neighbouring vertex with the highest edge weight that prefers v.
- Preferences and suitors are initialized to **nil**, i.e., no preference or suitor has been set.

F. Manne and M. Halappanavar, In: Proceedings IPDPS 2014, IEEE, pp. 519–528.

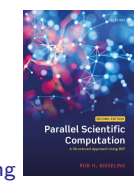


Only one preference and one suitor

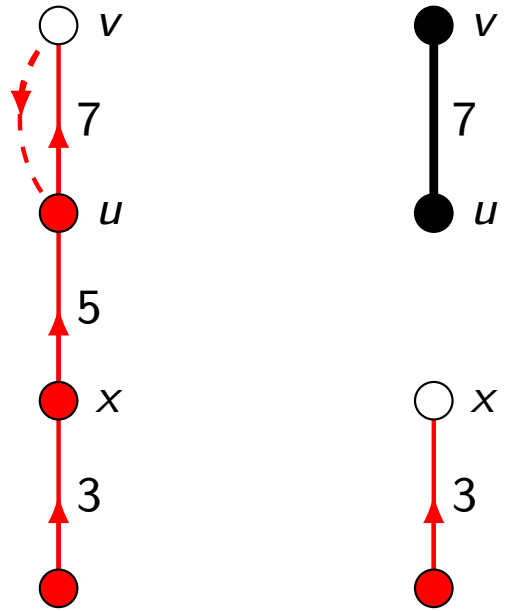
- A vertex can have only one preference and one suitor, since there are no ties.
- For each vertex v, it must hold that

$$\omega(v, \mathit{pref}(v)) \geq \omega(v, \mathit{suitor}(v)),$$

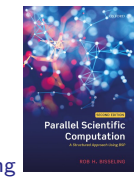
because otherwise the suitor of v would be a better candidate than the preferred choice of v.



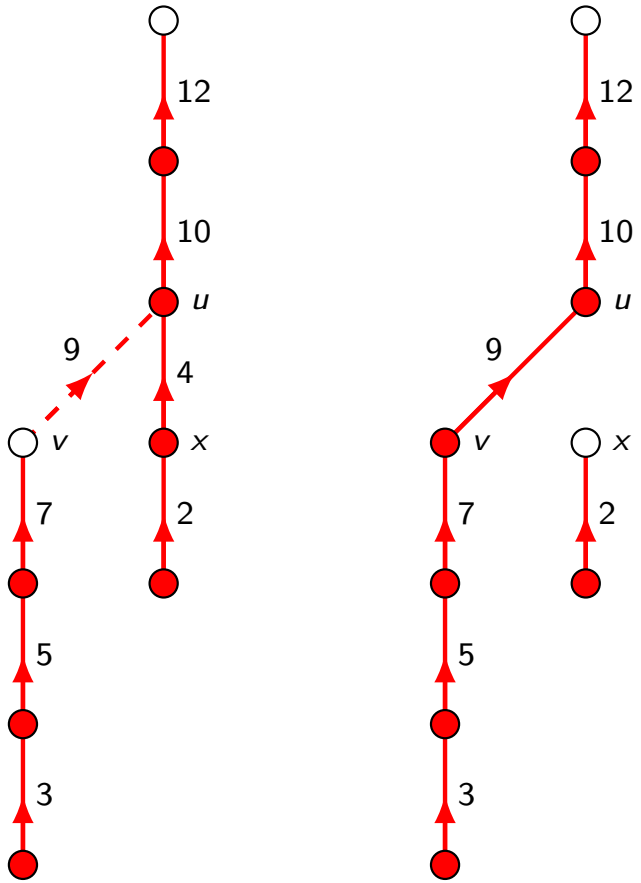
Setting a new, mutual preference



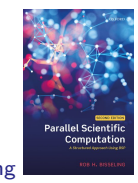
- Left (before setting): vertex v prefers u, where u already has set a preference for v.
- Right (afterwards): the preference is mutual and yields a match with weight 7. Vertex x loses its preference and must set a new preference.



Setting a new, nonmutual preference



- Vertex v prefers u, but u has a higher set preference.
- Vertex v becomes the suitor of u, replacing x.
- Vertex x loses its preference and must set a new preference.



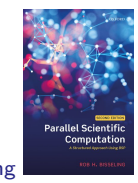
Finding the preference of a vertex

input: array Adj of length d, interval $[lo, hi]$, $0 \leq lo \leq hi < d$.
output: The preferred edge is returned and moved to the end.

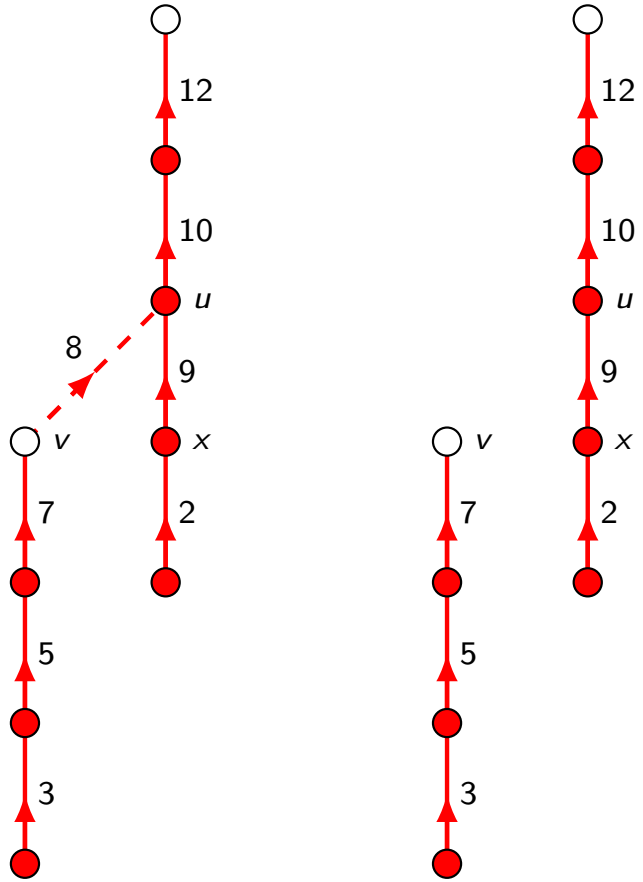
function FINDPREF(Adj, ω, lo, hi)

$\omega_{\max} := -\infty$;
for $i := lo$ **to** hi **do**
  if $\omega(e_{Adj[i]}) > \omega_{\max}$ **then**
    $i_{\max} := i$;
    $\omega_{\max} := \omega(e_{Adj[i]})$;
swap($Adj[i_{\max}], Adj[hi]$);
return $e_{Adj[hi]}$;

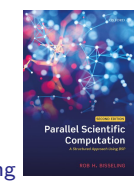
- The array Adj stores the adjacency list of a vertex, i.e., the indices k for which the edge e_k is connected to the vertex.
- The function simply searches for the heaviest edge and swaps its index into the end of the array.



Getting an immediate rejection



- Vertex v prefers u, where u already has a better suitor x.
- The edge (u, v) is dead and can be removed from the graph.



Finding the highest living edge while removing dead edges

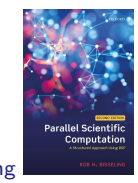
input: v: a vertex,
Adj: its adjacency list of length d_v,
...

output: The living edge $e_{Adj[i]}$ with highest index i.

function FindAlive($v, Adj, \omega, alive, suitor, d_v$)

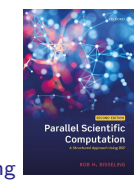
for $i := d_v - 1$ **to** 0 **step** -1 **do**
$(u, v) := e_{Adj[i]}$;
if $\omega(u, suitor(u)) > \omega(u, v)$ **then**
$alive(u, v) :=$ **false**;

if $alive(u, v)$ **then**
return;
else
$d_v := d_v - 1$;



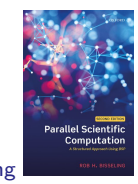
Data structure for the adjacency set

- The adjacency set Adj_v of vertex v can be stored as an array of length d_v representing the edges of v. This array is the adjacency list of v.
- Here, d_v is the degree of v, i.e., its number of neighbours.
- The adjacency list of a vertex corresponds to a matrix row in the compressed row storage (CRS) scheme for sparse symmetric matrices.
- Each edge (v_i, v_j) is represented twice, once corresponding to a nonzero a_{ij} in row i (the list of vertex v_i) and once corresponding to a_{ji} in row j (the list of vertex v_j).



Creating the adjacency list data structure

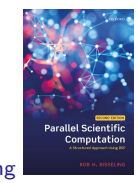
- The adjacency lists of all the vertices together can be stored in an array of length $2m$, where $m = |\mathcal{E}|$.
- This can be done by a counting sort in $\mathcal{O}(m+n)$ time for the whole graph, similar to creating a CRS data structure for a sparse matrix:
  - first, we count the number of edges of each vertex;
  - then, we determine the start and end of the space needed for each vertex;
  - finally, for each edge e_k, we place the index k twice, in the space corresponding to its two endpoints.



Local domination algorithm with suitors: initialization

for all $v \in \mathcal{V}$ **do**
    $suitor(v) := \textbf{nil}$;
for all $e \in \mathcal{E}$ **do**
    $alive(e) := \textbf{true}$;
$\mathcal{M} := \emptyset$;
$Q := \mathcal{V}$;

- $\mathcal{M}$ is the current set of matches.
- Q is the current set of unmatched but still matchable vertices without a preference.
- Q can be represented by a list of vertex numbers, either as a queue following the First In, First Out (FIFO) principle, or as a stack, following Last In, First Out (LIFO).
- For the sequential algorithm, we need not store the value of $pref(v)$ explicitly, as it can be derived from the *suitor* values. For the parallel algorithm, we need $pref(v)$.



Local domination algorithm with suitors: main loop

while $Q \neq \emptyset$ **do**

pick a vertex $v \in Q$;

$Q := Q \setminus \{v\}$;

$\mathrm{FindAlive}(v, Adj_v, \omega, alive, suitor, d_v)$;

{ Set preference for v }

if $d_v > 0$ **then**

$(u, v) := \mathrm{FindPref}(Adj_v, \omega, 0, d_v - 1)$;

$d_v := d_v - 1$;

$x := suitor(u)$;

if $x \neq$ **nil then**

$Q := Q \cup \{x\}$;

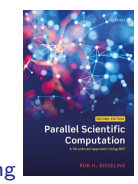
$alive(u, x) :=$ **false**;

$suitor(u) := v$;

if $u = suitor(v)$ **then**

$\mathcal{M} := \mathcal{M} \cup \{(u, v)\}$;

$d_v := 0$; $d_u := 0$;



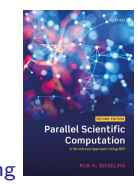
Improvement: partial sorting

- Partially sorting the adjacency lists by increasing weight helps find a preference quickly.
- To do this, we use splitters that split the array into smaller pieces, similar to the splitters of the quicksort algorithm.
- Here, we call an index r a splitter of the array x if

$$\begin{aligned} x_i &< x_r \quad \text{for all } i < r, \\ x_j &\geq x_r \quad \text{for all } j \geq r. \end{aligned}$$

- The maximum value x_j can then be found by searching only within the range $j \geq r$.
- We apply this to an array x defined by

$$x_i = \omega(e_{Adj_v[i]}), \quad \text{for } 0 \leq i < d_v.$$



Incorporating partial sorting

while $Q \neq \emptyset$ **do**

...

$\text{FindAlive}(v, Adj_v, \omega, alive, suitor, d)$;

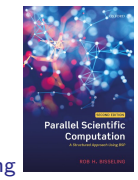
$r := \text{FindSplitter}(v, Adj_v, \omega, alive, splitter_v, suitor, d)$;

$(u, v) := \text{FindPref}(Adj_v, \omega, r, d_v - 1)$;

...

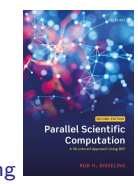
$\text{SplitAdj}(Adj_v, \omega, splitter_v, r, d_v - 1)$;

- The main loop of the algorithm calls a function FindSplitter, which is similar to FindAlive and returns a splitter r.
- We can now call FindPref with $lo = r$ instead of $lo = 0$.
- At the end of the main loop, a function SplitAdj is called, which is similar to the Split function of quicksort, see Section 1.8. It accelerates future searches for a preference.



Computation time of Algorithm 5.3

- The initializations of the main algorithm cost $\mathcal{O}(m+n)$.
- In each iteration of the main loop, either:
  - a vertex v is removed from Q and no vertex is put back, or
  - a vertex x is put back but then also a living edge (u, x) is killed.
- Therefore, the total number of iterations is at most $m+n$.
- The total computation time is $\mathcal{O}(m+n)$, excluding the cost of the partial sorting.

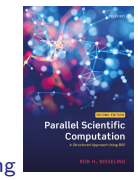


Computation time of the partial sorting

- The function SplitAdj chooses a random splitter and splits the data into a lower and upper part.
- We may have to split the upper part again, since we are interested in finding the highest weight. We only split the lower part if this becomes necessary later on.
- In the worst case, we perform a full quicksort for every vertex v, with an expected cost of $d_v \log_2 d_v$ operations.
- The expected total time for the fully sorted case is of order

$$\sum_v d_v \log_2 d_v \leq \sum_v d_v \log_2 \Delta = 2m \log_2 \Delta,$$

  where $\Delta = \max_v d_v$, the maximum vertex degree.
- Here, were we use that $\sum_v d_v = 2m$ (in this sum, every edge is counted twice).
- In the favourable case where we never split a lower part, the expected time for vertex v is $d_v + d_v/2 + \cdots + 1 \approx 2d_v$, and the total time is $4m$.



Summary

- A graph can be stored efficiently by a data structure consisting of adjacency lists. Each list stores the edges of one vertex. The total memory required for all n lists is $2m$.
- We have presented a matching algorithm based on local domination that repeatedly sets preferences for a vertex and matches a pair of vertices if their preferences are mutual.
- The suitor of a vertex v is the neighbouring vertex with the highest edge weight that prefers v.
- Storing the suitor allows for immediate rejection in case a vertex sets a preference worse than that of the current suitor.
- Partial sorting of adjacency lists based on a random splitter helps finding the heaviest edge quickly.