Lab 1 Solutions

I have seen many long and/or convoluted programs. This is normal: this may be your first exposure to parallel programming, and you were asked to solve a rather complicate problem. However, if your program was long and/or convoluted, or ran out of memory, or took forever to run, please read our solution.

(a) Prove that, on input $G = (V, E)$, the parallel algorithm terminates after at most $O(|V|^2)$ pulses.

Solution The idea of the proof is to bound the number of lifts and of saturating pushes, and then to prove that the number of pushes is related to the number of lifts. The latter relation can be proved by means of a "potential function" argument.

Consider a 'potential' function

$$\Phi = \max_{v \in X} h[v] \ ,$$

where $X \subseteq V$ is the set of overflowing nodes. If all the nodes manage to get rid of their excess during a push, then Φ decreases by at least one during that push. If some node preserves its excess, then Φ decreases or stays the same, and some node is lifted in the subsequent lift phase. If some node is lifted, then Φ increases, and some node is lifted by at least the amount Φ increases. Therefore we know that:

1. The total amount by which Φ increases is at most $2|V|^2$, since each node reaches height at most $2|V|$.
2. The number of pulses in which Φ stays the same is at most the number of lifts, that is $2|V|^2$.
3. Since $\Phi \geq 0$, the number of pulses in which Φ decreases is at most the total amount of which Φ increases, that is $2|V|^2$.

Therefore, after at most $O(|V|^2)$ pulses the algorithm terminates.

(b) Write a NESL function `ConvertInput` that transforms the input format into a 7-tuple `G = (degree, src, dst, mate, c, source, sink)`, where the first four components are as described above, `c` is the sequence of capacities, and `source` and `sink` are as in the input format.

Solution

```
function ConvertInput(edges, s, tee) =
let
    % add dummy zero-capacity edges %
    edges = edges ++ {((w, v), 0.0) : ((v, w), c) in edges};
    edges = {(edge, sum(capacities)) : (edge, capacities) in collect(edges) };

    % canonicalize the edge list, in ascending lexicographic order %
    rnk = rank({ b : ((a, b), c) in edges });
    edges = permute(edges, rnk);
    rnk = rank({ a : ((a, b), c) in edges });
    edges = permute(edges, rnk);

    % compute the 'mate' permutation.  mate->edge gives the
    other half of the edge.  This code is tricky: basically it suffices
    to compute the stable rank on the second coordinate of the edge.
    %
    mate = rank({b : ((a, b), c) in edges}) ;

    outgoing = int_collect({(a, b) : ((a, _), __) in edges; b in [0:#edges]});
    degree = {#b: (a, b) in outgoing};
    (extremes, capacities) = unzip(edges);
    (src, dst) = unzip(extremes);
in
    % return all we know about the graph %
    (degree, src, dst, mate, capacities, s, tee)
$
```

There are many things in the above code that may appear as hacks, but are actually in the spirit of data-parallel programming.

Adding the dummy edges This is done in two lines. First, add `((w, v), 0.0)` whenever `((v, w), c)` is in the graph. Second, collect all capacities (either real or dummy) belonging to the same directed edge, and sum them.

Sorting We want the edges in ascending lexicographical order, so that all edges originating from the same node are adjacent in the list. Solution: radix sort. We sort on the destination node, and then on the source. The best way to sort is to extract the key we are interested in, rank the keys, and permute. Many people tried to adapt quicksort to solve this problem: such approach is doomed to be slower and run out of memory.

Finding the mate Half a line of code `:-)`. Since the edges are already lexicographically sorted, it suffices to compute the stable rank of the `v`'s in the sequence of the edges (u, v). This is half a step of radix sort in reverse lexicographic order: we are not interested in performing the permutation, but only in knowing it. The idea is that if (u, v) would go to position p when the edges are sorted in reverse lexicographic order, then p is the position of the mate.

(c) Write `MaxDegreeNeighbor(G)` *without* using concurrent reads from the sequence of degrees.

Solution We stole this code from Carlo Maley and Ron Weiss:

```
function MaxDegreeNeighbor G =
let
    (degree, src, dst, mate, c, source, sink) = G;
    degreeList = flatten({ dist(x,x) : x in degree });
    neighbDegrees = degreeList->mate
in
    { max_val(v) : v in partition(neighbDegrees, degree) }
$
```

The idea is the following: each node *distributes* its degree to all its neighbors, then the value is sent to the neighbors, and finally each node computes the maximum of the values it received.

The given code is OK for the sequential implementation of NESL. On a really parallel implementation, it would probably be faster to use `permute(degreeList, mate)` instead of the read operation `degreeList->mate`. Reason: the read operation needs to go where the datum is, and bring it back, while `permute` sends the data to the right place in one step. This hack works because the `mate` permutation is the inverse of itself, and therefore you can either apply `mate` or its inverse.

(d) Write a function `MaxFlow(X)` that takes a graph `X` (in the input format), converts it into internal format and computes the maximum flow.

Solution

```
function InitialFlow(G) =
let
    (degree, src, dst, mate, C, s, tee) = G;
    flow = dist(0.0 , #src);
    NeighboursOfSource = { d : d in [0:#src]; src | src == s };
    FlowFromSource = { C : C; src | src == s };
```

```
    % set the flow on edges out of the source %
    flow = flow <- zip(NeighboursOfSource, FlowFromSource);

    % set the flow on edges into the source %
    flow = flow <- zip(mate->NeighboursOfSource, {-a : a in FlowFromSource})
in
    flow
$


%
Produce the initial height --- zero everywhere except the
source, which is labeled N = #V = #degree
%

function InitialHeight(G) =
let
    (degree, src, dst, mate, C, s, tee) = G;
    N = #degree;
    height = dist(0 , N)
in
    rep(height, N, s)
$

%
Compute the excess = sum of the incoming flow = -sum of the outgoing flow
%
function Excess(G, flow) =
let
    (degree, src, dst, mate, C, s, tee) = G;
    a = { - sum(p) : p in partition(flow, degree) };
in
    a <- [(s, 0.0), (tee, 0.0)]  % zero the excess of s and tee %
$


%
Compute the residual capacity of every edge
%
function ResidualCapacity(G, flow) =
let
```

```
    (degree, src, dst, mate, C, s, tee) = G;
in
    { c - flow : c; flow }
$

%
Do one pulse of the algorithm
%
function Pulse(G, flow, height) =
let
    (degree, src, dst, mate, C, s, tee) = G;
    Xsrc = Excess(G, flow)->src;
    R = ResidualCapacity(G, flow);
    srcl = height->src;
    dstl = height->dst;
    N = #degree;

    % compute the maximum push on each edge %
    MaxPush = { if (lv == lw + 1) and plusp(Rw) then Rw else 0.0 :
                lv in srcl; lw in dstl; Rw in R };

    % sigma contains the amount of flow pushed by all
      preceding edges in the list %
    sigma = flatten({ plus_scan(x) : x in partition(MaxPush, degree)});

    q = {  if (m + s >= ex) then ex - s  else m
         : m in MaxPush; s in sigma; ex in Xsrc };

    PushAmount = { if (s >= ex) then 0.0 else q
                   : q ; s in sigma; ex in Xsrc};

    % push flow %
    Nflow = { ef + p1 - p2 : p1 in PushAmount; p2 in PushAmount->mate;
              ef in flow};

    % compute new heights %
    R = ResidualCapacity(G, flow);
    tmp = { if (Rw <= 0.0) then (2 * N) else lw + 1 :
                lw in dstl; Rw in R };

    Nheight = { min_val(x) : x in partition(tmp, degree)}
              <- [(s, N), (tee, 0)]
```

```
in
    (Nflow, Nheight)
$

function Done(G, flow) =
let
    X = Excess(G, flow);
    epsilon = 1E-10;
in
    all({X < epsilon : X})
$

function ConvertOutput(G, flow) =
let
    (degree, src, dst, mate, C, s, tee) = G;

    % a contains the excess of all nodes %
    a = { - sum(p) : p in partition(flow, degree) };
    es = elt(a, s);
    et = elt(a, tee);
    Y = { ((u, v), ef) : u in src; v in dst; ef in flow }
in
    (Y, es, et)
$

function MaxFlowAux(G, flow, height) =
    if (Done(G, flow)) then
        flow
    else
        let
            (Nflow, Nheight) = Pulse(G, flow, height)
        in
            MaxFlowAux(G, Nflow, Nheight)
$

function MaxFlow(X) =
let
    G = ConvertInput(X);
    flow = MaxFlowAux(G, InitialFlow(G), InitialHeight(G))
in
    ConvertOutput(G, flow)
$
```

Most of the code is straightforward. We shall just give some explanations on the `Pulse` procedure.

First, we compute the maximum allowable push on each edge. This is the residual capacity `Rw` if it is greater than 0 (`plusp(Rw)`) and the source height is one plus the destination height; otherwise, the maximum push is 0.

Then, we must allocate the overflow among the outgoing edges. Our strategy is to allocate the maximum possible push amount to the edges that come earlier in the list. We do it in two steps. First, we compute the sum `sigma` of the maximum push on each edge that precedes the current one in the list of edges and has the same source node. This is essentially a segmented prefix:

```
sigma = flatten({ plus_scan(x) : x in partition(MaxPush, degree)});
```

Then, every node computes its own push amount, based on the knowledge of the excess and of the amount that preceding edges want to push.

Finally, the flow is pushed. This is done by adding the push amount to each edge, and subtracting the push amount of the mate.

We then compute all lifts. The idea is as follows: every edge computes a quantity (`tmp`) which is is one plus the height of the destination node if the edge belongs to the residual network, and infinite (actually, $2n$ suffices) otherwise. The new heights are then computed as the minimum value of `tmp` over the adjacent edges (plus some tweaking to maintain the source and sink to the right height).