Homework #4

(Due: May 12)

Task 1. [110 Points] The Truck that Couldn't.

This task will help a long-haul truck that once got stuck under an overpass. Now when going from one location to another the truck wants to know which roads to take so that it never again gets stuck under something somewhere along the route.

Figure 1: "The Truck that Couldn't" deliver the overpass because it was too heavy.

An entire road network will be given to you as a directed graph $\mathcal{G} = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ are locations, and each directed edge (v_i, v_j) is a direct road (a local road or a highway) going from location v_i to location v_j, where $i \neq j$ and $1 \leq i, j \leq n$. Each (v_i, v_j) is also labeled by a real-valued positive *height* $h(v_i, v_j)$ which gives the maximum height of a truck that can pass through that road (e.g., determined by the heights of overpasses and other hanging objects on the road). If $(v_i, v_j) \notin E$, $h(v_i, v_j)$ is assumed to have a value 0. The *height* value of a path is defined as the minimum of the h values of the edges in the path. For each pair $v_i, v_j \in V$, $\pi[i, j]$ is defined to be the largest of the height values of all paths going from v_i to v_j.

Figure 2 shows an iterative algorithm Loop-Max-Height for computing the $\pi[i, j]$ value for every pair of vertices $v_i, v_j \in V$. The input is an $n \times n$ matrix $\Pi[1 \ldots n, 1 \ldots n]$ in which for every pair

$i, j \in [1, n]$, $\Pi[i, j]$ is initialized with $h(v_i, v_j)$. After the algorithm terminates $\Pi[i, j] = \pi[i, j]$ for all $i, j \in [1, n]$.

In this task we will consider a number of recursive divide-and-conquer implementations of LOOP-MAX-HEIGHT which are shown in Figures 3, 5 and 6. For simplicity we will assume n to be a power of 2. Each of these implementations is launched by calling $\mathcal{A}(X, X, X)$, where X points to $\Pi[1 \ldots n, 1 \ldots n]$ initialized with edge lengths of G as described in the previous paragraph. In general, for each $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$, function $\mathcal{A}$ / $\mathcal{A}_{loop}$ accepts $X \equiv \Pi[i_1 \ldots i_2, j_1 \ldots j_2]$, $U \equiv \Pi[i_1 \ldots i_2, k_1 \ldots k_2]$ and $V \equiv \Pi[k_1 \ldots k_2, j_1 \ldots j_2]$ as inputs, where $i_1, i_2, j_1, j_2, k_1, k_2 \in [1, n]$ with $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 \geq 0$. Each update $\Pi[i, j] \leftarrow \max(\Pi[i, j], \min(\Pi[i, k], \Pi[k, j]))$ applied by $\mathcal{F}(X, U, V)$ / $\mathcal{F}_{loop}(X, U, V)$ updates $\Pi[i, j] \in X$ using $\Pi[i, k] \in U$ and $\Pi[k, j] \in V$, where $i_1 \leq i \leq i_2$, $j_1 \leq j \leq j_2$ and $k_1 \leq k \leq k_2$. In order to reduce the overhead of recursion each recursive function $\mathcal{F}(X, U, V)$ switches to an iterative function $\mathcal{F}_{loop}(X, U, V)$ as shown in Figure 4 when the problem size becomes small but still requires enough work to solve it so that the overhead of recursively reaching that problem size does not dominate the cost of solving the problem.

We want to avoid all data races in all subtasks below unless mentioned otherwise.

LOOP-MAX-HEIGHT(Π, n)

```
1. for k ← 1 to n do
2.     for i ← 1 to n do
3.         for j ← 1 to n do
4.             Π[i, j] ← max(Π[i, j], min(Π[i, k], Π[k, j]))
```

Figure 2: [ITERATIVE ALGORITHM] Looping code for computing $\pi[i, j]$ values.

$\mathcal{A}(X, U, V)$ $\{possibly\ X \equiv U \equiv V\}$

```
1. if X is an m × m matrix then A_loop(X, U, V)
   else
2.     A(X11, U11, V11)
3.     parallel: A(X12, U11, V12), A(X21, U21, V11)
4.     A(X22, U21, V12)
5.     A(X22, U22, V22)
6.     parallel: A(X21, U22, V21), A(X12, U12, V22)
7.     A(X11, U12, V21)
```

Figure 3: [RECURSIVE IMPLEMENTATION 1] The initial call is $\mathcal{A}(X, X, X)$, where X points to $\Pi[1 \ldots n, 1 \ldots n]$ and n is assumed to be a power of 2. By X_{11}, X_{12}, X_{21} and X_{22} we denote the top-left, top-right, bottom-left and bottom-right quadrant of X, respectively.

(*a*) [**10 Points**] Figure 2 shows a serial iterative algorithm LOOP-MAX-HEIGHT for computing the $\pi[i, j]$ value for every pair of vertices $v_i, v_j \in V$. Can you parallelize this algorithm by

$\mathcal{F}_{loop}(X, U, V)$ $\{\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}\}$

1. let $i_1, i_2, j_1, j_2, k_1, k_2 \in [1, n]$ with $i_2 - i_1 = j_2 - j_1 = k_2 - k_1 \geq 0$ be indices such that
   $X \equiv \Pi[i_1 \ldots i_2, j_1 \ldots j_2]$, $U \equiv \Pi[i_1 \ldots i_2, k_1 \ldots k_2]$ and $V \equiv \Pi[k_1 \ldots k_2, j_1 \ldots j_2]$.
2. ***for*** $k \leftarrow k_1$ ***to*** k_2 ***do***
3. ***for*** $i \leftarrow i_1$ ***to*** i_2 ***do***
4. ***for*** $j \leftarrow j_1$ ***to*** j_2 ***do***
5. $\Pi[i, j] \leftarrow \max(\Pi[i, j], \min(\Pi[i, k], \Pi[k, j]))$

Figure 4: Looping base case for the recursive implementation of Loop-Max-Height.

only replacing the serial ***for*** loops with parallel ***for*** loops? Why or why not? If you are able to parallelize, compute the work, span and parallelism of your parallel algorithm.

(b) [**10 Points**] If data races are allowed which serial ***for*** loops of Loop-Max-Height in Figure 2 can be replaced with parallel ***for*** loops without compromising the correctness of the algorithm? Justify your answer. If you are able to parallelize, compute the work, span and parallelism of your parallel algorithm.

(c) [**40 Points**] Assuming m to be a (small) constant[1] independent of n, compute the work, span and parallelism of each of the three recursive divide-and-conquer implementations of Loop-Max-Height given in Figures 3, 5 and 6.

(d) [**20 Points**] Observe that $\mathcal{A}_{loop}$, $\mathcal{B}_{loop}$, $\mathcal{C}_{loop}$, and $\mathcal{D}_{loop}$ must implement the same triply nested ***for*** loop as shown in Figure 4. Parallelize each of them using only parallel ***for*** loops. Compute the work, span and parallelism of each of those four functions.

(e) [**15 Points**] Consider the recursive implementation from Figure 6. Can you improve its parallelism even further by using extra space for storing intermediate values the way we did for recursive matrix multiplication? Show your analysis. How about the recursive implementations in Figures 3 and 5.

(f) [**15 Points**] Can you improve the parallelism of your parallel algorithm for $\mathcal{D}_{loop}$ from part 1(d) using extra space? Replace the serial ***for*** loop(s) you could not parallelize in part 1(d) with (possibly recursive) parallel code that uses extra space but do not change your parallel ***for*** loop(s). Analyze the work, span, parallelism and space usage of your improved parallel algorithm.

[1] for this subtask you may even assume $m = 1$ if you like

$\mathcal{A}(X, U, V)$ $\{possibly\ X \equiv U \equiv V\}$

1. ***if*** X is an $m \times m$ matrix ***then*** $\mathcal{A}_{loop}(X, U, V)$

   else

2. $\quad \mathcal{A}(X_{11}, U_{11}, V_{11})$
3. $\quad$ **parallel:** $\mathcal{B}(X_{12}, U_{11}, V_{12})$, $\mathcal{C}(X_{21}, U_{21}, V_{11})$
4. $\quad \mathcal{A}(X_{22}, U_{21}, V_{12})$
5. $\quad \mathcal{A}(X_{22}, U_{22}, V_{22})$
6. $\quad$ **parallel:** $\mathcal{B}(X_{21}, U_{22}, V_{21})$, $\mathcal{C}(X_{12}, U_{12}, V_{22})$
7. $\quad \mathcal{A}(X_{11}, U_{12}, V_{21})$

$\mathcal{B}(X, U, V)$ $\{X\ and\ U\ are\ dsjoint,\ but\ possibly\ X \equiv V\}$

1. ***if*** X is an $m \times m$ matrix ***then*** $\mathcal{B}_{loop}(X, U, V)$

   else

2. $\quad$ **parallel:** $\mathcal{B}(X_{11}, U_{11}, V_{11})$, $\mathcal{B}(X_{12}, U_{11}, V_{12})$
3. $\quad$ **parallel:** $\mathcal{B}(X_{21}, U_{21}, V_{11})$, $\mathcal{B}(X_{22}, U_{21}, V_{12})$
4. $\quad$ **parallel:** $\mathcal{B}(X_{21}, U_{22}, V_{21})$, $\mathcal{B}(X_{22}, U_{22}, V_{22})$
5. $\quad$ **parallel:** $\mathcal{B}(X_{11}, U_{12}, V_{21})$, $\mathcal{B}(X_{12}, U_{12}, V_{22})$

$\mathcal{C}(X, U, V)$ $\{X\ and\ V\ are\ dsjoint,\ but\ possibly\ X \equiv U\}$

1. ***if*** X is an $m \times m$ matrix ***then*** $\mathcal{C}_{loop}(X, U, V)$

   else

2. $\quad$ **parallel:** $\mathcal{C}(X_{11}, U_{11}, V_{11})$, $\mathcal{C}(X_{21}, U_{21}, V_{11})$
3. $\quad$ **parallel:** $\mathcal{C}(X_{12}, U_{11}, V_{12})$, $\mathcal{C}(X_{22}, U_{21}, V_{12})$
4. $\quad$ **parallel:** $\mathcal{C}(X_{12}, U_{12}, V_{22})$, $\mathcal{C}(X_{22}, U_{22}, V_{22})$
5. $\quad$ **parallel:** $\mathcal{C}(X_{11}, U_{12}, V_{21})$, $\mathcal{C}(X_{21}, U_{22}, V_{12})$

Figure 5: [Recursive Implementation 2] The initial call is $\mathcal{A}(X, X, X)$, where X points to $\Pi[1 \ldots n, 1 \ldots n]$ and n is assumed to be a power of 2. By X_{11}, X_{12}, X_{21} and X_{22} we denote the top-left, top-right, bottom-left and bottom-right quadrant of X, respectively.

$\mathcal{A}(X,U,V)$ $\{X \equiv U \equiv V\}$

1. ***if*** X is an $m \times m$ matrix ***then*** $\mathcal{A}_{loop}(X,U,V)$

   else

2. $\quad \mathcal{A}(X_{11},U_{11},V_{11})$
3. $\quad$ **parallel:** $\mathcal{B}(X_{12},U_{11},V_{12})$, $\mathcal{C}(X_{21},U_{21},V_{11})$
4. $\quad \mathcal{D}(X_{22},U_{21},V_{12})$
5. $\quad \mathcal{A}(X_{22},U_{22},V_{22})$
6. $\quad$ **parallel:** $\mathcal{B}(X_{21},U_{22},V_{21})$, $\mathcal{C}(X_{12},U_{12},V_{22})$
7. $\quad \mathcal{D}(X_{11},U_{12},V_{21})$

$\mathcal{B}(X,U,V)$ $\{X$ *and* U *are dsjoint, but* $X \equiv V\}$

1. ***if*** X is an $m \times m$ matrix ***then*** $\mathcal{B}_{loop}(X,U,V)$

   else

2. $\quad$ **parallel:** $\mathcal{B}(X_{11},U_{11},V_{11})$, $\mathcal{B}(X_{12},U_{11},V_{12})$
3. $\quad$ **parallel:** $\mathcal{D}(X_{21},U_{21},V_{11})$, $\mathcal{D}(X_{22},U_{21},V_{12})$
4. $\quad$ **parallel:** $\mathcal{B}(X_{21},U_{22},V_{21})$, $\mathcal{B}(X_{22},U_{22},V_{22})$
5. $\quad$ **parallel:** $\mathcal{D}(X_{11},U_{12},V_{21})$, $\mathcal{D}(X_{12},U_{12},V_{22})$

$\mathcal{C}(X,U,V)$ $\{X$ *and* V *are dsjoint, but* $X \equiv U\}$

1. ***if*** X is an $m \times m$ matrix ***then*** $\mathcal{C}_{loop}(X,U,V)$

   else

2. $\quad$ **parallel:** $\mathcal{C}(X_{11},U_{11},V_{11})$, $\mathcal{C}(X_{21},U_{21},V_{11})$
3. $\quad$ **parallel:** $\mathcal{D}(X_{12},U_{11},V_{12})$, $\mathcal{D}(X_{22},U_{21},V_{12})$
4. $\quad$ **parallel:** $\mathcal{C}(X_{12},U_{12},V_{22})$, $\mathcal{C}(X_{22},U_{22},V_{22})$
5. $\quad$ **parallel:** $\mathcal{D}(X_{11},U_{12},V_{21})$, $\mathcal{D}(X_{21},U_{22},V_{12})$

$\mathcal{D}(X,U,V)$ $\{X,$ U *and* V *are dsjoint}*

1. ***if*** X is an $m \times m$ matrix ***then*** $\mathcal{D}_{loop}(X,U,V)$

   else

2. $\quad$ **parallel:** $\mathcal{D}(X_{11},U_{11},V_{11})$, $\mathcal{D}(X_{12},U_{11},V_{12})$, $\mathcal{D}(X_{21},U_{21},V_{11})$, $\mathcal{D}(X_{22},U_{21},V_{12})$
3. $\quad$ **parallel:** $\mathcal{D}(X_{11},U_{12},V_{21})$, $\mathcal{D}(X_{12},U_{12},V_{22})$, $\mathcal{D}(X_{21},U_{22},V_{21})$, $\mathcal{D}(X_{22},U_{22},V_{22})$

Figure 6: [Recursive Implementation 3] The initial call is $\mathcal{A}(X,X,X)$, where X points to $\Pi[1\ldots n, 1\ldots n]$ and n is assumed to be a power of 2. By X_{11}, X_{12}, X_{21} and X_{22} we denote the top-left, top-right, bottom-left and bottom-right quadrant of X, respectively.

Task 2. [70 Points] Devices and Switches.

Consider m devices that need to be activated using n switches.

Each switch controls two *activation points* (AP) – one red and one green. If your turn a switch to point to its red AP it will activate all devices connected to that AP, otherwise it will activate all devices connected to its green AP. There is no restriction on how many (including zero) devices can be connected to an AP.

Each device is connected to either exactly one AP or exactly k different APs for some given integer $k > 1$. If a device is connected to exactly one AP it must be connected to a red AP. A device gets activated if at least one of the APs it is connected to becomes active, otherwise the device remains inactive.

Figure 7 shows an example.

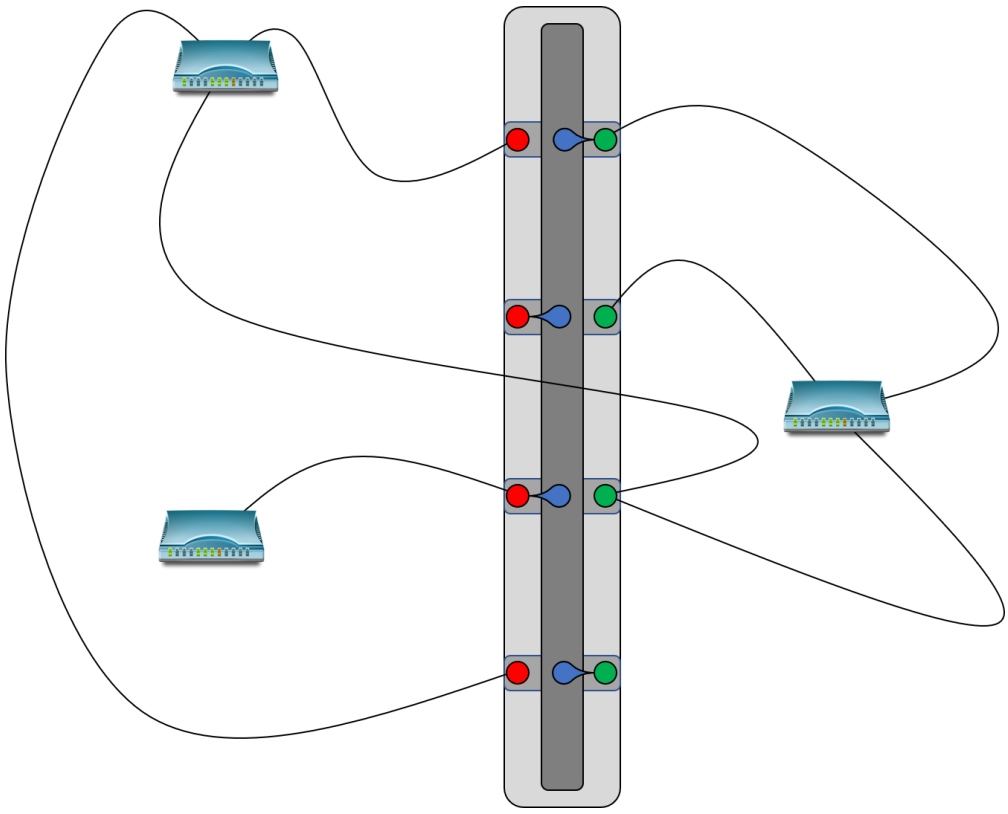

Figure 7: Three devices are connected to four switches but only two devices are active under the switch setting shown. If we turn the bottom switch to red then all three devices will be active.

Given m devices connected to the APs of n switches, we will analyze a very simple randomized approximation algorithm for finding a switch setting to activate a number of devices that is within a constant factor of the optimal. The algorithm is as follows: on every switch flip a biased coin that turns up *heads* with some probability $p > \frac{1}{2}$, and turn the switch to red if the coin turns up heads otherwise turn it to green.

(*a*) [**20 Points**] Prove that for $k = 2$ the algorithm above gives an expected $\min(p, 1 - p^2)$-approximation of the optimal.

(*b*) [**10 Points**] How do you turn the given algorithm to an expected $\frac{\sqrt{5}-1}{2}$-approximation algorithm for $k = 2$?

(*c*) [**10 Points**] Show that the algorithm from part 2(*b*) returns a $\frac{3}{5}$ or even better approximation of the optimal w.h.p. in m.

(*d*) [**30 Points**] Repeat parts 2(*a*)–2(*c*) for $k = 3$. What approximation bounds do you get?