PARALLEL-BFS(G, s, d)

(Inputs are an unweighted directed graph G with vertex set $G[V]$, and a source vertex $s \in G[V]$. For any vertex $u \in G[V]$, $\Gamma(u)$ denotes the set of vertices adjacent to u. The output will be returned in d, where for each $u \in G[V]$, $d[u]$ will be set to the shortest distance (i.e., number of edges on the shortest path) from s to u.)

1. ***parallel for*** each $u \in G[V]$ ***do*** $d[u] \leftarrow +\infty$ {*initialize all distances to* $+\infty$}
2. $d[s] \leftarrow 0$ {*the source vertex is at distance* 0}
3. $p \leftarrow$ #processing cores
4. $Q^{in} \leftarrow$ collection of p empty FIFO queues $Q^{in}.q[1]$, $Q^{in}.q[2]$, ..., $Q^{in}.q[p]$ {Q^{in} *will hold vertices in the current BFS level*}
5. $Q^{out} \leftarrow$ collection of p empty FIFO queues $Q^{out}.q[1]$, $Q^{out}.q[2]$, ..., $Q^{out}.q[p]$ {*vertices in the next BFS level generated from* Q^{in} *will be stored in* Q^{out}}
6. $S \leftarrow$ collection of p segment pointers (global) {*for* $1 \le i \le p$, $S[i]$ *will point to the queue segment currently being explored by thread* i}
7. $Q^{in}.q[1].enque(\ s\ )$ {*start with BFS level 0 by enqueueing the source vertex*}
8. ***while*** $Q^{in} \neq \emptyset$ ***do*** {*iterate until* Q^{in} *(i.e., the current BFS level) is empty*}
9. ***parallel for*** $i = 1$ ***to*** p ***do*** $S[i] \leftarrow$ entire $Q^{in}.q[i]$ as a single segment {*thread* i *will start with* $Q^{in}.q[i]$}
10. ***for*** $i = 1$ ***to*** $p - 1$ ***do*** {*from vertices in* Q^{in} *generate vertices*
11. ***spawn*** PARALLEL-BFS-THREAD(i, G, $Q^{out}.q[i]$, d) *in the next BFS level by launching* $p - 1$
12. PARALLEL-BFS-THREAD(p, G, $Q^{out}.q[p]$, d) *threads concurrent to the current thread*}
13. ***sync*** {*wait until all vertices in* Q^{in} *have been processed*}
14. $Q^{in} \Leftrightarrow Q^{out}$ {*swap the roles of* Q^{in} *and* Q^{out}}
15. $Q^{out} \leftarrow \emptyset$ {*empty the queues in* Q^{out}}

Figure 2: Parallel breadth-first search (BFS) on a graph.

PARALLEL-BFS-THREAD(i, G, Q^o, d)

(Inputs are the id $i \in [1, p]$ of the current thread, and an unweighted directed graph G with vertex set $G[V]$. For any vertex $u \in G[V]$, $\Gamma(u)$ denotes the set of vertices adjacent to u. For each such vertex v the correct BFS level will be stored in $d[v]$. All vertices in the next level of BFS discovered by the current thread will be put in the output queue Q^o which is a single queue used exclusively by the current thread. We assume that $S[1:p]$ is a globally accessible queue segment identifiers, where $S[i]$ keeps track of the input queue segment currently being explored by thread i.)

1. ***while*** ($S[i] \neq \emptyset$) ***do***
2. ***while*** ($S[i] \neq \emptyset$) ***do*** {*while the input segment is not empty*}
3. $u \leftarrow S[i].extract(\ )$ {*extract the next vertex from the segment*}
4. ***for*** each $v \in \Gamma(u)$ ***do*** {*consider each vertex adjacent to* u}
5. ***if*** $d[v] \leftarrow +\infty$ ***then*** {*if that adjacent vertex* v *has not yet been visited*}
6. $d[v] \leftarrow d[u] + 1$ {*distance to* v *is* 1 *more than that to* u}
7. $Q^o.enque(\ v\ )$ {*enqueue* v *in the output queue for future exploration*}
8. $t \leftarrow 0$ {*count number of steal attempts*}
9. ***while*** ($S[i] = \emptyset$) ***and*** ($t <$ MAX-STEAL-ATTEMPTS) ***do*** {*try to steal* $\leq$ MAX-STEAL-ATTEMPTS *times*}
10. $r \leftarrow$ RAND(1, p) {*pick a random victim*}
11. ***if*** TRY-LOCK(r) ***then*** {*if able to secure exclusive access to the victim*}
12. ***if*** ($|S[r]| >$ MIN-STEAL-SIZE) ***then*** {*if victim's current queue segment is not too small*}
13. $S[i] \leftarrow$ second half of $S[r]$ {*steal second half of work from victim's segment*}
14. UNLOCK(r) {*give up exclusive access*}
15. $t \leftarrow t + 1$ {*done with one more steal attempt*}

Figure 3: Parallel breadth-first search (BFS) on a graph.