Homework #1

(Due: March 12)

SERIAL-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 sortest distance (i.e., number of edges on the sortest path) from s to u.)

1. ***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 distannce* 0}
3. $Q \leftarrow \emptyset$ {*start with an empty FIFO queue* Q}
4. $Q.enque(\ s\ )$ {*enqueue the source vertex*}
5. ***while*** $Q \neq \emptyset$ ***do*** {*iterate until the queue becomes empty*}
6. $\quad u \leftarrow Q.deque(\ )$ {*dequeue the first vertex* u *from the queue*}
7. $\quad$ ***for*** each $v \in \Gamma(u)$ ***do*** {*consider each vertex adjacent to* u}
8. $\quad\quad$ ***if*** $d[v] = +\infty$ ***then*** {*if that adjacent vertex* v *has not yet been visited*}
9. $\quad\quad\quad d[v] \leftarrow d[u] + 1$ {*distance to* v *is* 1 *more than that to* u}
10. $\quad\quad\quad Q.enque(\ v\ )$ {*enqueue* v *for future exploration*}

Figure 1: Serial breadth-first search (BFS) on a graph.

Task 1. [200 Points] Parallel BFS with Work Stealing.

(*a*) **[10 Points]** Implement the serial BFS algorithm (SERIAL-BFS) given in Figure 1.

(*b*) **[5 Points]** Consider the parallel BFS algorithm (PARALLEL-BFS) given in Figure 2. Explain how the algorithm may give rise to race conditions, and why the output will still be correct.

(*c*) **[15 Points]** Show that in any given iteration of the ***while*** loop in PARALLEL-BFS, the same vertex may appear multiple times in Q^{in}, but not more than once in any given queue of Q^{in}. Modify the algorithm so that exactly one of those multiple instances of the same vertex is expanded in lines 4–7 of PARALLEL-BFS-THREAD. You are allowed to use only $\mathcal{O}(1)$ additional time per instance for this modification. Explain how and why your modification works. Also prove that the same vertex cannot appear in Q^{in} in two different iterations of the ***while*** loop.

(*d*) **[20 Points]** Let q_l be the total number of entries in all input queues (i.e., $Q_{in}.q[i]$'s) at the start of the exploration of a particular BFS level l. Prove that w.h.p. in p, after the first $\Theta\left(p \log p \log\left(\frac{q_l}{\text{MIN-STEAL-SIZE}}\right)\right)$ steal attempts in the entire system, all steal attempts will fail.

(*e*) **[10 Points]** Explain why $cp \log p$ is a good choice for MAX-STEAL-ATTEMPTS, where $c > 1$ is a constant.

(*f*) [**30 Points**] Observe that in any given BFS level each thread is either waiting to be launched, or exploring vertices, or trying to steal, or waiting at the ***sync*** point. Now for an input graph G of diameter D with n vertices, m edges and maximum degree Δ, use the observation above to (upper) bound the parallel running time T_p of the entire algorithm. Assume that MAX-STEAL-ATTEMPTS $= \Theta(p \log p)$, and the algorithm has been modified to avoid the exploration of duplicate vertices (following part (*c*)). Give a lower bound on the size of the input graph which makes the algorithm work-optimal.

(*g*) [**15 Points**] Modify the algorithm so that Δ disappears from T_p while other terms remain unchanged. Such modifications are useful for handling scale-free graphs[1] more efficiently which arise frequently in real-world scenarios (e.g., the web graph, social network graphs, biological interaction networks, etc.).

(*h*) [**5 Points**] Consider lines 10–13 of PARALLEL-BFS (Figure 2), and suggest modifications to reduce the time a thread waits to be launched. How does that change the bounds you proved in parts (*f*) and (*g*)?

(*i*) [**35 Points**] Implement the algorithms from parts (*f*) and (*g*) without the modification for avoiding duplicate exploration. Run on a machine with at least 8 cores (same for parts (*j*) and (*k*)), and empirically find and report the best value for MIN-STEAL-SIZE and also for MAX-STEAL-ATTEMPTS. Optimize your code as much as possible.

(*j*) [**35 Points**] Create a table that compares the running times of your serial implementation from part (*a*) and the parallel implementations from part (*i*) using all cores. For each input file (in Appendix 1) create a separate row in the table showing the running times of all three algorithms as well as the speedup factors of the two parallel implementations w.r.t. (*a*).

(*k*) [**20 Points**] Generate a `Cilkview` strong scalability plot (see slides 15–18 of lecture 5) for each of the two parallel implementations from part (*i*) using the wikipedia graph (see Appendix 1) as input.

Task 2. [Optional, No Point] Lockfree Parallel BFS.

(*a*) [**No Point**] Observe that in lines 9–17 of PARALLEL-BFS-THREAD (Figure 3) the thief uses a lock to make its own queue unavailable to other thieves while it is trying to steal, and uses another lock to restrict access to its victim's queue segment while a steal is in progress. Can you eliminate all locks from this algorithm without making it incorrect? You are not allowed to use any atomic instructions either. Prove the correctness of your algorithm, and also that it terminates.

(*b*) [**No Point**] Implement and optimize your algorithm from part (*a*), and repeat parts (*j*) and (*k*) of Task 1 with this implementation.

[1] graphs in which vertex degrees follow the power law degree distribution

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 $Q^{in}.q[1],\ Q^{in}.q[2],\ \ldots,\ Q^{in}.q[p]$ {Q^{in} *will hold vertices in the current BFS level*}
5. $Q^{out} \leftarrow$ collection of p empty FIFO queues $Q^{out}.q[1],\ Q^{out}.q[2],\ \ldots,\ Q^{out}.q[p]$ {*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*** (TRUE) ***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. LOCK(i) {*lock self until done with stealing*}
10. ***while*** ($S[i] = \emptyset$) ***and*** ($t <$ MAX-STEAL-ATTEMPTS) ***do*** {*try to steal* $\leq$ MAX-STEAL-ATTEMPTS *times*}
11. $r \leftarrow$ RAND($1,\ p$) {*pick a random victim*}
12. ***if*** TRY-LOCK(r) ***then*** {*if able to secure exclusive access to the victim*}
13. ***if*** ($|S[r]| >$ MIN-STEAL-SIZE) ***then*** {*if victim's current queue segment is not too small*}
14. $S[i] \leftarrow$ second half of $S[r]$ {*steal second half of work from victim's segment*}
15. UNLOCK(r) {*give up exclusive access*}
16. $t \leftarrow t + 1$ {*done with one more steal attempt*}
17. UNLOCK(i) {*make self available to thieves*}
18. ***if*** ($S[i] = \emptyset$) ***then break*** {*terminate thread if all steal attempts fail*}

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

APPENDIX 1: Input/Output Format

Your code must read from standard input and write to standard output.

- **Input Format:** The first line of the input will contain three integers giving the number of vertices (n), number of edges (m), and the number of source vertices (r), respectively. Each of the next m lines will contain two integers u and v ($1 \leq u, v \leq n$) denoting a directed edge from vertex u to vertex v. The edges will be sorted in nondecreasing order of the first vertex. Each of the next r lines will contain one integer s ($1 \leq s \leq n$) giving the index of a source vertex.
- **Output Format:** The output will consist of r lines – one for each source vertex. Line i ($1 \leq i \leq r$) will contain two indegers d_i and c_i, where d_i is the maximum BFS level of a vertex from the i-th source vertex given in the input file, and c_i is the checksum value as computed by the following function.

```
// Computes a very simple checksum by adding all d values.
// When some d[ i ] = infinity, replaces that infinity with nVertices
// during checksum calculation, i.e., assumes that d[ i ] = nVertices.
unsigned long long computeChecksum( void )
{
   cilk::reducer_opadd< unsigned long long > chksum;

   cilk_for ( int i = 0; i < nVertices; i++ )
      chksum += d[ i ];

   return chksum.get_value( );
}
```

- **Sample Input/Output:** `/work/01905/rezaul/CSE638/HW1/samples` on Lonestar.
- **Test Input/Output (see Table 1):** `/work/01905/rezaul/CSE638/HW1/turn-in` on Lonestar.

Graph	Description	n	m
cage15	DNA electrophoresis, 15 monomers in polymer	5.2M	99.2M
cage14	DNA electrophoresis, 14 monomers in polymer	15.1M	27.1M
freescale	Large circuit, Freescale Semiconductor	3.4M	18.9M
Wikipedia	Gleich/wikipedia-20070206	3.6M	45M
kkt-power	Optimal power flow, nonlinear optimization (KKT)	2M	8.1M
RMAT100M	RMAT Graph generated using Graph-500 RMAT generator	10M	100M
RMAT1B	RMAT Graph generated using Graph-500 RMAT generator	10M	1B

Table 1: Input graphs with #vertices (n) and #edges (m). Note that $1M = 10^6$ and $1B = 10^9$.

APPENDIX 2: What to Turn in

One compressed archive file (e.g., zip, tar.gz) containing the following items.

- Source code, makefiles and job scripts.
- A PDF document containing all answers and plots.
- Output generated for the input files in `/work/01905/rezaul/CSE638/HW1/turn-in/` on Lonestar. If the name of the input file is `xxxxx-in.txt`, please name the output files as `xxxxx-1a-out.txt`, `xxxxx-1f-out.txt`, `xxxxx-1g-out.txt` and `xxxxx-2b-out.txt` for tasks $1(a)$, $1(f)$, $1(g)$ and $2(b)$, respectively.
- Output files generated by Cilkview.

APPENDIX 3: Things to Remember

- **Please never run anything that takes more than a minute or uses multiple cores on TACC login nodes.** TACC policy strictly prohibits such usage. They reserve the right to suspend your account if you do so. All runs must be submitted as jobs to compute nodes (even when you use Cilkview or PAPI).
- Please store all data in your work folder ($WORK), and not in your home folder ($HOME).
- When measuring running times please exclude the time needed for reading the input and writing the output. Measure only the time needed by the algorithm. Do the same thing when you run Cilkview.
- Please make sure that speedup values for trial runs (or measured speedups) are included in the Cilkview plots you generate.