CSE613: Parallel Programming, Spring 2012 **Date:** March 31

Homework #2

(Due: April 14)

```
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.     $u \leftarrow Q.deque( )$                               {dequeue the first vertex $u$ from the queue}
 7.     for each $v \in \Gamma(u)$ do                           {consider each vertex adjacent to $u$}
 8.         if $d[v] = +\infty$ then                            {if that adjacent vertex $v$ has not yet been visited}
 9.             $d[v] \leftarrow d[u] + 1$                      {distance to $v$ is 1 more than that to $u$}
10.             $Q.enque( v )$                                  {enqueue $v$ for future exploration}
```

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

Task 1. [135 Points] Parallel BFS with Split Queues.

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

(*b*) [**25 Points**] Implement the parallel BFS algorithm (PARALLEL-BFS) given in Figure 2. Optimize your code as much as possible based on parts (*d*), (*e*) and (*f*).

(*c*) [**5 Points**] Explain how PARALLEL-BFS may give rise to race conditions. Also explain why the output will still be correct.

(*d*) [**15 Points**] The *Q.nextSegment*( ) function shown in Figure 3 locks a global data structure, and thus serializes all accesses to it. Explain how you will implement the function (particularly lines 4 and 6–7), and what value you will use for n_{seg} so that in any given iteration of the ***while*** loop in PARALLEL-BFS the total time spent inside *Q.nextSegment*( ) by all threads is only $\mathcal{O}\left(\frac{|Q^{in}|}{p}\right)$.

(*e*) [**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 in Figure 2 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, and $\mathcal{O}(1)$ additional space per vertex for this modification. Explain how and why your modification works as well as any (reasonable) support you need from the hardware for your approach to work.

(*f*) [**10 Points**] Optimize the space used by the queues in Q^{in} and Q^{out}. Initially in lines 4 and 5 of PARALLEL-BFS allocate a constant amount of space per queue (e.g., space for 256 vertices). Then during each enqueue operation check if there is space in the queue for adding the new vertex, and if not, double the size of the queue. Also just before returning from PARALLEL-BFS-THREAD check if at least 25% of Q^o is full, and if not, contract its size to two times the number of elements currently in it (but not below the initial size of the queue). When implementing these in `cilk++` you need to use the "Miser" memory manager for thread safety and efficiency (chapter 10 of *Intel Cilk++ SDK Programmer's Guide* explains how to use Miser).

(*g*) [**10 Points**] Analyze the work and span of your implementation of PARALLEL-BFS in part (*a*) with all modifications outlined in parts (*d*), (*e*) and (*f*).

(*h*) [**20 Points**] Instead of using the centralized mechanism as shown in $Q.nextSegment(\,)$ (see Figure 3) for distributing work from Q^{in} to all threads, implement the following distributed work-stealing mechanism. For $1 \leq i \leq p$, thread i starts with queue $Q^{in}.q[i]$ as its input queue. However, whenever a thread becomes idle, it chooses a random thread, and steals half (2nd half) of the remaining vertices from that thread's queue. For efficiency you need to make sure that stealing does not involve explicit copying of the stolen vertices to a separate location.

(*i*) [**15 Points**] Create a table that compares the running times of your serial implementation in part (*a*), and the parallel implementations in parts (*b*) and (*h*) using all cores. For each input file (in Appendix 2) create a separate row in the table showing the running times of (*a*), (*b*) and (*h*) as well as the speedup factors of (*b*) and (*h*) w.r.t. (*a*).

(*j*) [**10 Points**] Identify the input file on which your serial implementation in part (*a*) takes the longest time not exceeding 10 minutes. Use that input file to generate a `Cilkview` scalability plot (see slides 32–33 of lecture 3) for part (*b*) and one for part (*h*).

Task 2. [Optional, No Points] Further Analysis of Parallel BFS with Split Queues.

(*a*) [**No Points**] Analyze the number of attempted steals and the number of successful steals (both probabilistic and worst-case bounds) for the BFS implementation in part (*h*).

```
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 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 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. $Q^{in}.q[1].enque( s )$                      {start with BFS level 0 by enqueueing the source vertex}
 7. while $Q^{in} \neq \emptyset$ do              {iterate until $Q^{in}$ (i.e., the current BFS level) is empty}
 8.     $n_{seg} \leftarrow$ #segments to split $Q^{in}$ into (within $\Theta(1)$ factor) for concurrent processing
 9.     $Q^{in}.s_{seg} \leftarrow \left\lceil \frac{|Q^{in}|}{n_{seg}} \right\rceil$     {elements in $Q^{in}$ will be split into segments of size $\leq Q^{in}.s_{seg}$}
10.     for $i = 1$ to $p - 1$ do                                 {from vertices in $Q^{in}$ generate vertices
11.         spawn PARALLEL-BFS-THREAD( $G$, $Q^{in}$, $Q^{out}.q[i]$, $d$ )    in the next BFS level by launching $p - 1$
12.     PARALLEL-BFS-THREAD( $G$, $Q^{in}$, $Q^{out}.q[p]$, $d$ )             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 $Q^{out}$}
```

```
PARALLEL-BFS-THREAD( $G$, $Q^i$, $Q^o$, $d$ )
(Inputs are an unweighted directed graph $G$ with vertex set $G[V]$, and a collection $Q^i$ of queues (shared by all threads)
containing the input vertices for this level of BFS. 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 (as opposed to $Q^i$) used exclusively by the current
thread. For each such vertex $v$ the correct BFS level will be stored in $d[v]$ by the current thread. For any vertex
$u \in G[V]$, $\Gamma(u)$ denotes the set of vertices adjacent to $u$.)
 1. while ( $S = Q^i.nextSegment( )$ ) $\neq \emptyset$ do       {extract a segment of vertices from $Q^i$}
 2.     while $S \neq \emptyset$ do                             {consider all vertices in this segment}
 3.         $u \leftarrow S.extract( )$                          {extract any vertex $u$ from the seqment}
 4.         for each $v \in \Gamma(u)$ do                         {consider each vertex adjacent to $u$}
 5.             if $d[v] = +\infty$ 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}
```

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

```
$Q.nextSegment(\ )$
($Q$ is a collection of queues $Q.q[1],\ Q.q[2],\ \ldots,\ Q.q[t]$, for some $t > 0$. This function returns a segment of at most
$Q.s_{seg}$ queued elements from one of $Q$'s nonempty queues.)
  1. Lock( )
  2. $S \leftarrow \emptyset$
  3. if $|Q| > 0$ then                                   {if at least one queue in $Q$ is nonempty}
  4.     $i \leftarrow$ smallest index such that $Q.q[i] \neq \emptyset$
  5.     $k \leftarrow \min(\ Q.s_{seg},\ |Q.q[i]|\ )$
  6.     for $j \leftarrow 1$ to $k$ do
  7.         $S \leftarrow S \cup Q.q[i].deque(\ )$
  8. Unlock( )
  9. return $S$
```

Figure 3: Get a segment of queued elements from a queue in the collection Q.

APPENDIX 1: Input/Output Format for Task 1

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:** Folder `/work/01905/rezaul/CSE613/HW2/samples` on Lonestar.

APPENDIX 2: What to Turn in

Please email one compressed archive file (e.g., zip, tar.gz) containing the following items to `cse613hw@cs.stonybrook.edu`.

– Source code, makefiles and job scripts for Task 1.

– A PDF document containing all answers and plots.

– Output generated for the input files in `/work/01905/rezaul/CSE613/HW2/samples/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-1b-out.txt` and `xxxxx-1h-out.txt` for tasks $1(a)$, $1(b)$ and $1(h)$, respectively. No need to generate an output if it takes more than an hour or overflows the RAM. In such cases simply state the cause of failure in the output file.

– 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.