HW5: Graph Biconnectivity

Course:	ENEE651/CMSC751
Title:	Graph Biconnectivity
Date Assigned:	May 1, 2016
Date Due:	May 15, 2016, 11:59pm
Contact:	Ankit Mondal – amondal2@umd.edu

1 Assignment Goal

Identify the biconnected components for an undirected graph $G = (V, E)$ using the **Tarjan-Vishkin biconnectivity algorithm**, described below.

2 Problem Statement

Definitions. Biconnectivity is a property of undirected graphs; an undirected graph G is called *biconnected* if and only if it is connected and remains so after removing any vertex and all edges incident on that vertex. A graph S is an *induced subgraph* of G if it comprises a subset of the vertices of G and all the edges of G connecting two vertices in S. A *biconnected component* of G is an induced subgraph of G that is biconnected whose vertex set cannot be expanded while maintaining the biconnectivity of its induced subgraph. A vertex whose removal increases the number of connected components in the graph is called an *articulation point*.

Given an undirected graph $G = (V, E)$, the Tarjan-Vishkin biconnectivity algorithm identifies the biconnected components of G by identifying the connected components of an auxiliary graph $G' = (V', E')$ whose vertices are the edges of G (i.e., $V' = E$). The biconnected components of G define a partitioning of the edges of G: every edge in E belongs to exactly one biconnected component of G. Implement an algorithm to identify for each edge in G the biconnected component to which it belongs.

2.1 Parallel Algorithm Description (Tarjan-Vishkin)

The Tarjan-Vishkin biconnectivity algorithm proceeds as follows:

1. Compute a spanning tree T of the input graph.
2. Root the spanning tree and find an Euler tour of the rooted spanning tree.
3. Using the Euler tour, determine the preorder number of each vertex and the size of the subtree rooted at each vertex.
4. For each vertex v, search the subtree $T(v)$ rooted at v for the vertex with the lowest preorder number in $T(v)$ or reachable from a vertex in $T(v)$ by a non-tree edge of G. Do the same for the highest preorder number. Call the results $low(v)$ and $high(v)$, respectively.

5. Construct the auxiliary graph $G'' = (V'', E'')$, where V'' is the set of tree edges of G, by adding edges between pairs of vertices in V'' based on their low and high numbers using the following rules ($p(v)$ is the parent of v):

   (a) For each edge v, w in $G - T$, add $\{\{p(v), v\}, \{p(w), w\}\}$ to G'' if and only if v and w are unrelated in T (i.e., neither v nor w is an ancestor of the other).

   (b) For each edge $v \rightarrow w$ in T, with v the parent of w, add $\{\{p(v), v\}, \{v, w\}\}$ to G'' if and only if $low(w) < v$ or $high(w) \geq v + size(v)$.

6. Compute the connected components of G''. Two tree edges in G are in the same biconnected component if and only they are in the same connected component of G''.

7. For each non-tree edge, identify the biconnected component to which it belongs by looking at the endpoint with the higher preorder number and assigning the edge to the same biconnected component as the tree edge connecting that endpoint to its parent in the spanning tree.

2.2 Serial Algorithm Description (Hopcroft-Tarjan)

Perform a depth-first search (DFS) of G starting from any vertex, maintaining the following information:

- $num(v)$: the order of v in the DFS traversal of G (the first vertex visited is 0, the second is 1, and so on)
- $low(v)$: the lowest *num* among the descendants of v (including v itself) and the vertices reachable from a descendant of v via a back edge
- *estack*: a stack of the edges traversed (both tree edges and back edges) so far; upon returning to an articulation point u after visiting one of its children v, the edges on *estack* from $u \rightarrow v$ to the top of *estack* represent the biconnected component of G containing the edge $u \rightarrow v$

The algorithm proceeds as follows:

1. Advance to the starting vertex r and set $num(r) := 0$ and $low(r) := 0$.

2. Upon advancing to a vertex v, examine each edge e incident on it except for the edge leading back to its parent. There are three cases:

   (a) $e = (v, w)$ is a tree edge (w has not yet been visited): push e onto *estack* and assign w the first number that has not yet been assigned to any vertex. Also, set $low(w) := num(w)$. Enter w.

   (b) $e = (v, u)$ is a back edge (u has already been visited and $num(u) < num(v)$): push e onto *estack* and set $low(v) := min(low(v), num(u))$. Do not advance to u (since u has already been visited).

   (c) $e = (v, u)$ is a forward edge (u has already been visited, but $num(u) > num(v)$): ignore e.

3. When returning from a vertex w that was reached from v via the edge $e = (v, w)$, set $low(v) := min(low(v), low(w))$. Also, check whether $low(w) < num(v)$. If not, then v is an articulation point, and all the edges from the top of *estack* down to and including e (and their antiparallel copies) form a single biconnected component; pop them from the stack.

4. When returning from r, the edges remaining on the stack (and their antiparallel copies) form a single biconnected component.

3 Assignment

1. Implement the parallel algorithm for generating the list of biconnected components using the **Tarjan-Vishkin biconnectivity** algorithm. Name your code file biconnectivity.p.c.
2. Implement a serial algorithm for generating the list of biconnected components using a serial biconnectivity algorithm of your choice. Name your code file biconnectivity.s.c.

Important: Both parallel and serial algorithms must be as efficient as possible. You will be graded based on both correctness and efficiency of your implementations.

4 Input

4.1 Setting up the environment

To get the source file templates and the *Makefile* for compiling programs, log in to your account in the class server and extract the *biconnectivity.tgz* using the following command:

```
$ tar xzvf /opt/xmt/class/xmtdata/biconnectivity.tgz
```

This will create the directory *biconnectivity* which contains the C file templates that you are supposed to edit and a Makefile.

As opposed to the previous assignments, data files are not included in this package. Instead they are located under */opt/xmt/class/xmtdata/biconnectivity*. The Makefile system will automatically use the appropriate data files following your make commands so you don't need to make a copy of them in your work environment. You have the OS privileges to view the header files however you cannot edit them.

4.2 Input format

The type and size of the data structures provided is given in the following table.

#define N	The number of vertices in the graph
#define M	The number of edges in the graph (each edge counts twice)
int edges[M][2]	The start and end vertex of each edge
int antiparallel[M]	The index in the `edges` array of the antiparallel copy of each edge
int vertices[N]	The index in the `edges` array at which point the edges incident to vertex begin
int degrees[N]	The degree of each vertex
int bcc[M]	Result array: an integer identifying the biconnected component to which each edge belongs

Declaration of temporary/auxiliary arrays: You can declare any number of global arrays and variables in your program as needed. For example, this is valid XMTC code:

```
#define N 16384

int temp1[16384];
int temp2[2*N];
int pointer;

int main() {
 //...
}
```

4.3 Data sets

The following data sets are provided:

Dataset	N	M	Header file	Binary File
graph0	11	28	$DATA/graph0/biconnectivity.h	$DATA/graph0/biconnectivity.xbo
graph1	49	372	$DATA/graph1/biconnectivity.h	$DATA/graph1/biconnectivity.xbo
graph2	9473	73844	$DATA/graph2/biconnectivity.h	$DATA/graph2/biconnectivity.xbo
graph3	13098	473212	$DATA/graph3/biconnectivity.h	$DATA/graph3/biconnectivity.xbo

$DATA is */opt/xmt/class/xmtdata/biconnectivity*. Note that each edge is listed twice in the input file. For example, the undirected graph0 has only 14 edges.

Graph example. We have provided an example graph given by the dataset `graph0` for debugging and exemplification purposes. This corresponds to the graph in Figure 1.

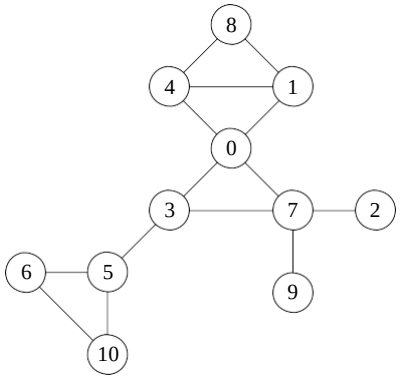


Figure 1: Graph example: `graph0` dataset.

5 Library routines

To assist you in implementing the parallel algorithm, the following set of C functions has been provided. The functions are implemented in parallel and until last year, could only be called from serial code (not within a spawn block). Since there have been many updates in the XMT FPGA, you might want to try calling these functions from even within spawn blocks.

5.1 Graph connectivity (with or without spanning tree)

- Prototype:

```
typedef int edge_t[2];
void connectivity(edge_t edges[], int n, int m, int D[]);
void connectivityTG(edge_t edges[], int antiparallel[], int n, int m, int D[], int T[]);
```

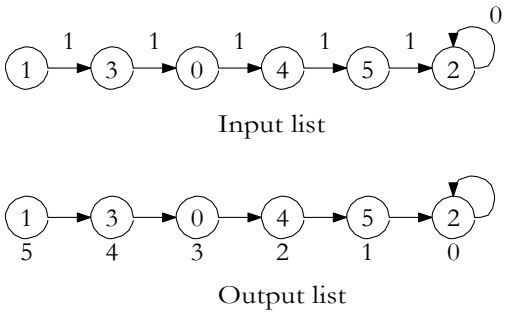
- Input:

- *edges*[*m*]: array of (undirected) edges
- (only for `connectivityTG`) *antiparallel*[*m*]: array of indices of antiparallel copies of edges
- *n*: number of vertices
- *m*: number of (undirected) edges

• Output:

- *D*[*n*]: array of numbers identifying the connected component in which each vertex lies; *D*[*u*] = *D*[*v*] if and only if *u* and *v* are in the same connected component
- (only for `connectivityTG`) *T*[*m*]: array of flags indicating whether each edge is in the generated spanning tree of the input graph; *T*[*i*] = 1 indicates that *edges*[*i*] is in the spanning tree, *T*[*i*] = 0 indicates that it is not.

5.2 List ranking



i	D[i]	R[i]
0	4	1
1	3	1
2	2	0
3	0	1
4	5	1
5	2	1

Input to list_rank

i	D[i]	R[i]
0	-	3
1	-	5
2	-	0
3	-	4
4	-	2
5	-	1

Output of list_rank

Figure 2: List ranking example.

• Prototype:

```
void list_rank(int D[], int R[], int n, int head);
```

• Input:

- *D*[*n*]: successor of each element; *D*[*tail*] = *tail* if *tail* is the last element
- *R*[*n*]: distance to the next element; *R*[*tail*] = 0 if *tail* is the last element
- *n*: number of list elements
- *head*: index of the first element in the list (the element with no predecessor); in Figure 2, *head* = 1

• Output:

- *R*[*n*]: distance from each element to the last element in the list

5.3 Summation, Minimum, and Maximum

- Prototype:

```
int sum_int(int a[], int n);
int min_int(int a[], int n);
int max_int(int a[], int n);
```

- Input:
  - *a*[*n*]: array to find the sum/min/max of
  - *n*: size of array
- Output:
  - Returns the sum/min/max of *a*[]

5.4 Segmented sum

Note: The sum[] array must be initialized to the identity for the given operation. For sum and max, this is 1. For min, this is the largest possible element (use 0x7fffffff if you do not know). Note: Elements in the same segment must occur consecutively in the input array.

- Prototype:

```
void segmented_sum_int(int a[], int seg[], int n, int sum[]);
void segmented_min_int(int a[], int seg[], int n, int sum[]);
void segmented_max_int(int a[], int seg[], int n, int sum[]);
```

- Input:
  - *a*[*n*]: array to find the segmented sum/min/max of
  - *seg*[*n*]: an array of integers uniquely identifying the segment to which each element of *a*[] belongs; *seg*[*i*] = *seg*[*j*] if and only if *a*[*i*] and *a*[*j*] are in the same segment
  - *n*: size of array
- Output:
  - *sum*[]: *sum*[*k*] receives the sum of all elements *a*[*i*] with *seg*[*i*] == *k*

5.5 Prefix sum

- Prototype:

```
void prefix_sum_int(int a[], int n, int sum[]);
void prefix_min_int(int a[], int n, int sum[]);
void prefix_max_int(int a[], int n, int sum[]);
```

- Input:
  - *a*[*n*]: array to find the prefix sum/min/max of

- *n*: size of array

- Output:
  - *sum*[*n*]: prefix sum/min/max of *a*[] (i.e., *sum*[*i*] = sum/min/max of *a*[0]…*a*[*i*])

5.6 Range minimum

Note: To use these functions, you should declare the input array to have enough extra space to hold the auxiliary data generated by the preprocessing stage. Specifically, if the input array *a*[] holds *n* elements, you should declare it with a size of at least $2n + \lceil \log_2 n \rceil$. The extra elements should not be used by your code; they will be filled in by preprocess_range_min_int and used by query_range_min_int. For this project, it will suffice to declare *a* with a size of 3*n*.

5.6.1 Preprocess

- Prototype:

```
void preprocess_range_min_int(int a[], int n);
void preprocess_range_max_int(int a[], int n);
```

- Input:
  - *a*[*n*]: array to preprocess
  - *n*: size of array

- Output:
  - *a*[$2n + \lceil \log_2 n \rceil$]: balanced binary tree of minimum/maximum values; the nodes of the tree are stored consecutively in the array, level by level, beginning with the leaves

5.6.2 Query

Note: this is a serial routine and may be called from within a spawn block by any number of threads

- Prototype:

```
int query_range_min_int(int aux[], int n, int begin, int end);
int query_range_max_int(int aux[], int n, int begin, int end);
```

- Input:
  - *aux*[$2n + \lceil \log_2 n \rceil$]: preprocessed array
  - *n*: size of original array, not including elements added by the preprocessing step
  - *begin*: index of first element in the range to query
  - *end*: index of last element in the range to query

- Output:
  - Returns the minimum/maximum of *aux*[*begin*…*end*]

6 Hints and remarks

1. Use the output of `connectivityTG` to create a spanning tree in the same format as the input data.
2. List ranking is time consuming, so make as few calls to `list_rank` as possible.
3. Perform the computation of *low* and *high* in two steps:
   (a) Compute local values for *low* and *high* for each vertex based only on the non-tree edges incident on that vertex.
   (b) Use the Euler tour of the spanning tree to reorder the local *low* and *high* arrays such that range min/max queries over the arrays can be used to identify the low and high numbers for a subtree rooted at a given vertex.
4. When using `connectivity`, you only need one copy of each undirected edge. If you add both an edge and its antiparallel counterpart, the output will still be correct, but your code will be slower.

7 Compiling and executing via the Makefile system

You can use the provided makefile system to compile and run your programs. For the smallest data set (graph0) parallel connectivity program is run as follows:

```
> make run INPUT=biconnectivity.p.c DATA=graph0
```

This command will compile and run the biconnectivity.p.c program with the graph0 data set. For other programs and data sets, change the name of the input file and the data set. You can use the `make check` command to compile and run your program and check the result for correctness.

```
> make check INPUT=biconnectivity.p.c DATA=graph0
```

If you need to just compile the input file (no run):

```
> make compile INPUT=biconnectivity.p.c DATA=graph0
```

You can get help on available commands with

```
> make help
```

Note that, you can still use the `xmtcc` and `xmtfpga` commands as in the earlier assignments. You can run with the makefile system first to see the commands and copy them to command line to run manually.

8 Grading Criteria and Submission

In this assignment, you will be graded on the correctness of your programs (biconnectivity.s.c and biconnectivity.p.c) and the performance of your parallel implementation in terms of completion time. Performance of your program will be compared against the performance of our reference implementation on graph2 and graph3 data sets. The cycle count for the reference implementation on graph2 is 5.8M cycles and on graph3 it is 25.4M cycles. For graph2 this corresponds to a speedup factor of 1.8 over our serial implementation of the connectivity algorithm. For graph3 the speedup factor is 2.3. You should create a table named table.txt to record the observed clock cycles, as was done in the previous assignments.

The correctness of your code will be checked via the `make check` rule as mentioned in the previous section. Your output is first normalized so that the biconnected components are numbered consecutively starting with 0. The normalized output is then compared to the correct output stored in the same directory as the input data files for the corresponding graph. You have to store your results in the bcc global variable (see Section 4.2) in order for the check to be performed correctly.

Please note that you are not required to provide any analysis of time and work complexities for this assignment.

Please remove any printf statements that you may have placed for debugging purposes from your code as they will affect the performance and possibly break the automated grading script. Once you have the two source files ready you can check the correctness of your programs[1]:

```
> make testall
```

check the validity of your submission:

```
> make submitcheck
```

and submit:

```
> make submit
```

[1] `make testall` command runs all possible combinations of programs and data files. Since it blocks the FPGA for a long time, use this command judiciously in order not to inconvenience other users.